

Python Cookbook 3rd Edition Documentation

Contents:

Copyright

书名：《Python Cookbook》3rd Edition
作者：David Beazley, Brian K. Jones
译者：熊能
版本：第3版
出版社：O'Reilly Media, Inc.
出版日期：2013年5月08日
Copyright © 2013 David Beazley and Brian Jones. All rights reserved.

更多发布信息请参考 <http://oreilly.com/catalog/errata.csp?isbn=9781449340377>

前言

项目主页

<https://github.com/yidao620c/python3-cookbook>

译者的话

人生苦短，我用Python！

译者一直坚持使用Python3，因为它代表了Python的未来。虽然向后兼容是它的硬伤，但是这个局面迟早会改变的，而且Python3的未来需要每个人的帮助和支持。目前市面上的教程书籍，网上的手册大部分基本都是2.x系列的，专门基于3.x系列的书籍少的可怜。

最近看到一本《Python Cookbook》3rd Edition，完全基于Python3，写的也很不错。为了Python3的普及，我也不自量力，想做点什么事情。于是乎，就有了翻译这本书的冲动了！这不是一项轻松的工作，却是一件值得做的工作：不仅方便了别人，而且对自己翻译能力也是一种锻炼和提升。

译者会坚持对自己每一句的翻译负责，力求高质量。但受能力限制，也难免有疏漏或者表意不当的地方。如果译文中有什么错漏的地方请大家见谅，也欢迎大家随时指正：

vidao620@gmail.com

作者的话

自从2008年以来，Python3横空出世并慢慢进化。Python3的流行一直被认为需要很长一段时间。事实上，到我写这本书的2013年，绝大部分的Python程序员仍然在生产环境中使用的是版本2系列，最主要是因为Python3不向后兼容。毫无疑问，对于工作在遗留代码上的每个程序员来讲，向后兼容是不得不考虑的问题。但是放眼未来，你就会发现Python3给你带来不一样的惊喜。

正如Python3代表未来一样，新的《Python Cookbook》版本相比较之前的版本有了一个全新的改变。最重要的是，这个意味着本书是一本非常前沿的参考书。书中所有代码都是在Python3.3版本下面编写和测试的，并没有考虑之前老版本的兼容性，也没有标注旧版本下的解决方案。这样子可能会有争议，但是我们最终的目的是写一本完全基于最新最先进工具和语言的书籍。希望这本书能成为在Python3下编码和想升级之前遗留代码的程序员们的优秀教程。

毫无疑问，编写一本这样的书会冒一定的编辑风险。如果在网上搜索Python教程的话，会搜到很多很多。比如ActiveState's Python recipes或者Stack Overflow，但是绝大部分都已经是过时的了。这些教程除了是基于Python2编写之外，可能还有很多解决方案在不同的版本之间是不一样的(比如2.3和2.4版本)。另外，它们还会经常使用一些过时的技术，这些已经内置到Python3.3里面去了。寻找完全基于Python3的教程真的难上加难啊。

这本书的所有主题都是基于已经存在的代码和技术，而不是专门去寻找Python3特有的教程。在原有代码基础上，我们完全使用最新的Python技术去改造。所以，任何想使用最新技术编写代码的程序员，都可以将本书当做一本很好的参考书籍。

在讨论的主题选择方面，我们不可能囊括Python领域所有的东西。因此，我们优先选择了Python语言核心部分，以及一些在开发中常见的问题和任务。另外，这里讨论的很多技术都是Python 3最新才出现的，所以如果工作在Python老版本下，即便是最有经验的程序员可能之前也没见过这些东西。另外，这些示例程序也会偏向于展示一些有用的编程技术(比如设计模式)，而不是仅仅定位在一些具体的问题上。尽管也提及到了有一些第三方包，但是本书主要定位在Python语言核心和标准库。

这本书适合谁

这本书的目标读者是那些想深入理解Python语言机制和最新编程技能的资深程序员。很多讨论都是标准库，框架和应用程序使用到的高级技术。本书所有示例均假设读者已经有了一定的编程背景并且可以很容易的读懂相关主题(比如基本的计算机科学知识，数据

结构知识，算法复杂度，系统编程，并行，C语言编程等)。另外，每个示例都只是一个入门指导，如果读者想深入研究，需要自己去查阅更多资料。因此，我们假定读者可以很熟练的使用搜索引擎以及知道怎样查询在线的Python文档。

这本书不适合Python的初学者。事实上，本书已经假定了读者已经有了一定的Python基础，看完过几本入门书籍。本书也不是那种快速参考手册(可以很快的查询某个模块下的某个函数)。本书旨在聚焦几个最重要的主题，演示几种可能的解决方案，作为一个跳板，你可以经此进入一些更高级的主题，这些可以在网上或者参考手册中找到。

本书示例代码

本书几乎所有源代码均可以在 <http://github.com/dabeaz/python-cookbook> 上面找到。作者欢迎各位修正bug，改进代码和评论。

本书就是帮助你完成你的工作。一般来讲，只要在本书上面的实例代码，你都可以随时拿过去在你的源码和文档中使用。你不需要向我们申请许可，除非你抄袭的太过分了。比如说复制几个代码片段去完成一个程序是不需要许可的，贩卖或者分发实例代码的光盘也不需要许可，引用本书和实例代码去网上回答一个问题也不需要许可。但是，合并大量的代码带你的正式产品或文档中去必须得到我们的许可。

我们不会要求你添加代码的出处，包括标题，作者，出版社，ISBN。比如：Python Cookbook, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7. 但是如果你这么做了，我们会很感激的。

联系我们

请将关于本书的评论和问题发送给出版社：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

本书网站: http://oreil.ly/python_cookbook_3e，上面有勘误表，示例和一些其他信息。

如果想要评论或者是问一下本书技术方面的问题，请发送邮件至：

bookquestions@oreilly.com

更多关于我们的书籍，讨论会，新闻，请访问我们的网站: <http://www.oreilly.com>

在Facebook上查找我们: <http://facebook.com/oreilly>

在Twitter上关注我们: <http://twitter.com/oreillymedia>

在YouTube上观看我们: <http://www.youtube.com/oreillymedia>

感谢

我们由衷的感谢本书的技术审核者Jake Vanderplas, Robert Kern 和 Andrea Crotti的非常有用的评论和建议，还有Python社区的帮助和鼓励。我们还想感谢上一个版本的编辑 Jake Vanderplas, Robert Kern, and Andrea Crotti。尽管这个版本是最新的，但是前一个版本已经提供了一个感兴趣主题和解决方案的框架。最后，最重要的就是，我们要感谢所有预览版本的读者，他们的评论和改进意见对本书很有帮助。

第一章：数据结构和算法

Python提供了大量的内置数据结构，包括列表，集合以及字典。大多数情况下使用这些数据结构是很简单的。但是，我们也会经常碰到诸如查询，排序和过滤等等这些普遍存在的问题。因此，这一章的目的就是讨论这些比较常见的问题和算法。另外，我们也会给出在集合模块 `collections` 当中操作这些数据结构的方法。

Contents:

1.1 解压序列赋值给多个变量

问题

现在有一个包含N个元素的元组或者是序列，怎样将它里面的值解压后同时赋值给N个变量？

解决方案

任何的序列(或者是可迭代对象)可以通过一个简单的赋值语句解压并赋值给多个变量。唯一的前提就是变量的数量必须跟序列元素的数量是一样的。

代码示例:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
'ACME'
>>> date
(2012, 12, 21)
>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>
```

如果变量个数和序列元素的个数不匹配，会产生一个异常。

代码示例：

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

讨论

实际上，这种解压赋值可以用在任何可迭代对象上面，而不仅仅是列表或者元组。包括字符串，文件对象，迭代器和生成器。

代码示例：

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

有时候，你可能只想解压一部分，丢弃其他的值。对于这种情况Python并没有提供特殊的语法。但是你可以使用任意变量名去占位，到时候丢掉这些变量就行了。

代码示例：

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

你必须保证你选用的那些占位变量名在其他地方没被使用到。

1.2 解压可迭代对象赋值给多个变量

问题

如果一个可迭代对象的元素个数超过变量个数时，会抛出一个 `ValueError`。那么怎样才能从这个可迭代对象中解压出N个元素出来？

解决方案

Python的星号表达式可以用来解决这个问题。比如，你在学习一门课程，在学期末的时候，你想统计下家庭作业的平均成绩，但是排除掉第一个和最后一个分数。如果只有四个分数，你可能就直接去简单的手动赋值，但如果有的24个呢？这时候星号表达式就派上用场了：

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

另外一种情况，假设你现在有一些用户的记录列表，每条记录包含一个名字、邮件，接着就是不确定数量的电话号码。你可以像下面这样分解这些记录：

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

值得注意的是上面解压出的 `phone_numbers` 变量永远都是列表类型，不管解压的电话号码数量是多少(包括0个)。所以，任何使用到 `phone_numbers` 变量的代码就不需要做多余的类型检查去确认它是否是列表类型了。

星号表达式也能用在列表的开始部分。比如，你有一个公司前8个月销售数据的序列，但是你想看下最近一个月数据和前面7个月的平均值的对比。你可以这样做：

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

下面是在Python解释器中执行的结果：

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

讨论

扩展的迭代解压语法是专门为解压不确定个数或任意个数元素的可迭代对象而设计的。通常，这些可迭代对象的元素结构有确定的规则（比如第1个元素后面都是电话号码），星号表达式让开发人员可以很容易的利用这些规则来解压出元素来。而不是通过一些比较复杂的手段去获取这些关联的元素值。

值得注意的是，星号表达式在迭代元素为可变长元组的序列时是很有用的。比如，下面是一个带有标签的元组序列：

```
records = [
    ('foo', 1, 2),
    ('bar', 'hello'),
    ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

星号解压语法在字符串操作的时候也会很有用，比如字符串的分割。

代码示例：

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

有时候，你想解压一些元素后丢弃它们，你不能简单就使用 `*`，但是你可以使用一个普通的废弃名称，比如 `_` 或者 `ign`。

代码示例：


```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *__, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

在很多函数式语言中，星号解压语法跟列表处理有许多相似之处。比如，如果你有一个列表，你可以很容易的将它分割成前后两部分：

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

如果你够聪明的话，还能用这种分割语法去巧妙的实现递归算法。比如：

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

然后，由于语言层面的限制，递归并不是Python擅长的。因此，最后那个递归演示仅仅是个好奇的探索罢了，对这个不要太认真了。

1.3 保留最后N个元素

问题

在迭代操作或者其他操作的时候，怎样只保留最后有限几个元素的历史记录？

解决方案

保留有限历史记录正是 `collections.deque` 大显身手的时候。比如，下面的代码在多行上面做简单的文本匹配，并只返回在前N行中匹配成功的行：

```
from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for li in lines:
        if pattern in li:
            yield li, previous_lines
        previous_lines.append(li)

# Example use on a file
if __name__ == '__main__':
    with open(r'../../cookbook/somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('-' * 20)
```

讨论

我们在写查询元素的代码时，通常会使用包含 `yield` 表达式的生成器函数，也就是我们上面示例代码中的那样。这样可以将搜索过程代码和使用搜索结果代码解耦。如果你还不清楚什么是生成器，请参看4.3节。

使用 `deque(maxlen=N)` 构造函数会新建一个固定大小的队列。当新的元素加入并且这个队列已满的时候，最老的元素会自动被移除掉。

代码示例：

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

尽管你也可以手动在一个列表上实现这一的操作(比如增加、删除等等)。但是这里的队列方案会更加优雅并且运行得更快些。

更一般的, `deque` 类可以被用在任何你只需要一个简单队列数据结构的场合。如果你不设置最大队列大小, 那么就会得到一个无限大小队列, 你可以在队列的两端执行添加和弹出元素的操作。

代码示例:

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

在队列两端插入或删除元素时间复杂度都是 $O(1)$, 而在列表的开头插入或删除元素的时间复杂度为 $O(N)$ 。

1.4 查找最大或最小的N个元素

问题

怎样从一个集合中获得最大或者最小的N个元素列表?

解决方案

`heapq`模块有两个函数: `nlargest()` 和 `nsmallest()` 可以完美解决这个问题。

```
import heapq
nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
```

两个函数都能接受一个关键字参数，用于更复杂的数据结构中：

```
portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]
cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

译者注：上面代码在对每个元素进行对比的时候，会以 `price` 的值进行比较。

讨论

如果你想在集合中查找最小或最大的N个元素，并且N小于集合元素数量，那么这些函数提供了很好的性能。因为在底层实现里面，首先会先将集合数据进行堆排序后放入一个列表中：

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heapq.heapify(nums)
>>> nums
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

堆数据结构最重要的特征是 `heap[0]` 永远是最小的元素。并且剩余的元素可以很容易的通过调用 `heapq.heappop()` 方法得到，该方法会先将第一个元素弹出来，然后用下一个最小的元素来取代被弹出元素(这种操作时间复杂度仅仅是 $O(N)$ ，N是堆大小)。比如，如果想要查找最小的3个元素，你可以这样做：

```
>>> heapq.heappop(nums)
-4
>>> heapq.heappop(nums)
1
>>> heapq.heappop(nums)
2
```

当要查找的元素个数相对比较小的时候，函数 `nlargest()` 和 `nsmallest()` 是很合适的。如果你仅仅想查找唯一的最小或最大($N=1$)的元素的话，那么使用`min()`和`max()`函数会更快些。类似的，如果 N 的大小和集合大小接近的时候，通常先排序这个集合然后再使用切片操作会更快点 (`sorted(items)[:N]` 或者是 `sorted(items)[-N:]`)。需要在正确场合使用函数`nlargest()` 和 `nsmallest()`才能发挥它们的优势(如果 N 快接近集合大小了，那么使用排序操作会更好些)。

尽管你没有必要一定使用这里的方法，但是堆数据结构的实现是一个很有趣并且值得你深入学习的东西。基本上只要是数据结构和算法书籍里面都会有提及到。`heapq` 模块的官方文档里面也详细的介绍了堆数据结构底层的实现细节。

1.5 实现一个优先级队列

问题

怎样实现一个按优先级排序的队列？并且在这个队列上面每次`pop`操作总是返回优先级最高的那个元素

解决方案

下面的类利用 `heapq` 模块实现了一个简单的优先级队列：

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

下面是它的使用方式：

```

>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>

```

仔细观察可以发现，第一个 `pop()` 操作返回优先级最高的元素。另外注意到如果两个有着相同优先级的元素(`foo` 和 `grok`)，`pop`操作按照它们被插入到队列的顺序返回的。

讨论

这一小节我们主要关注 `heapq` 模块的使用。函数 `heapq.heappush()` 和 `heapq.heappop()` 分别在队列 `_queue` 上插入和删除第一个元素，并且队列 `_queue` 保证第一个元素拥有最小优先级(1.4节已经讨论过这个问题)。`heappop()` 函数总是返回“最小的”的元素，这就是保证队列 `pop` 操作返回正确元素的关键。另外，由于 `push` 和 `pop` 操作时间复杂度为 $O(N)$ ，其中 N 是堆的大小，因此就算是 N 很大的时候它们运行速度也依旧很快。

在上面代码中，队列包含了一个 `(-priority, index, item)` 的元组。优先级为负数的目的是使得元素按照优先级从高到低排序。这个跟普通的按优先级从低到高排序的堆排序恰巧相反。

`index` 变量的作用是保证同等优先级元素的正确排序。通过保存一个不断增加的 `index` 下标变量，可以确保元素安装它们插入的顺序排序。而且，`index` 变量也在相同优先级元素比较的时候起到重要作用。

为了阐明这些，先假定 `Item` 实例是不支持排序的：

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

如果你使用元组 `(priority, item)`，只要两个元素的优先级不同就能比较。但是如果两个元素优先级一样的话，那么比较操作就会跟之前一样出错：

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

通过引入另外的 `index` 变量组成三元组 `(priority, index, item)`，就能很好的避免上面的错误，因为不可能有两个元素有相同的 `index` 值。Python在做元组比较时候，如果前面的比较以及可以确定结果了，后面的比较操作就不会发生了：

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

如果你想在多个线程中使用同一个队列，那么你需要增加适当的锁和信号量机制。可以查看12.3小节的例子演示是怎样做的。

`heapq` 模块的官方文档有更详细的例子程序以及对于堆理论及其实现的详细说明。

1.6 字典中的键映射多个值

问题

怎样实现一个键对应多个值的字典(也叫 `multidict`)?

解决方案

一个字典就是一个键对应一个单值的映射。如果你想要一个键映射多个值，那么你就需要将这多个值放到另外的容器中，比如列表或者集合里面。比如，你可以像下面这样构造这样的字典：

```
d = {
    'a' : [1, 2, 3],
    'b' : [4, 5]
}
e = {
    'a' : {1, 2, 3},
    'b' : {4, 5}
}
```

选择使用列表还是集合取决于你的实际需求。如果你想保持元素的插入顺序就应该使用列表，如果想去掉重复元素就使用集合（并且不关心元素的顺序问题）。

你可以很方便的使用 `collections` 模块中的 `defaultdict` 来构造这样的字典。

`defaultdict` 的一个特征是它会自动初始化每个 `key` 刚开始对应的值，所以你只需要关注添加元素操作了。比如：

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
```

需要注意的是，`defaultdict` 会自动为将要访问的键(就算目前字典中并不存在这样的键)创建映射实体。如果你并不需要这样的特性，你可以在一个普通的字典上使用

`setdefault()` 方法来代替。比如：


```
d = {} # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
```

但是很多程序员觉得 `setdefault()` 用起来有点别扭。因为每次调用都得创建一个新的初始值的实例(例子程序中的空列表`[]`)。

讨论

一般来讲，创建一个多值映射字典是很简单的。但是，如果你选择自己实现的话，那么对于值的初始化可能会有点麻烦，你可能会像下面这样来实现：

```
d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
    d[key].append(value)
```

如果使用 `defaultdict` 的话代码就更加简洁了：

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

这一小节所讨论的问题跟数据处理中的记录归类问题有大的关联。可以参考1.15小节的例子。

1.7 字典排序

问题

你想创建一个字典，并且在迭代或序列化这个字典的时候能够控制元素的顺序。

解决方案

为了能控制一个字典中元素的顺序，你可以使用 `collections` 模块中的 `OrderedDict` 类。在迭代操作的时候它会保持元素被插入时的顺序，示例如下：

```
from collections import OrderedDict
def ordered_dict():
    d = OrderedDict()
    d['foo'] = 1
    d['bar'] = 2
    d['spam'] = 3
    d['grok'] = 4
    # Outputs "foo 1", "bar 2", "spam 3", "grok 4"
    for key in d:
        print(key, d[key])
```

当你想要构建一个将来需要序列化或编码成其他格式的映射的时候，`OrderedDict` 是非常有用的。比如，你想精确控制以JSON编码后字段的顺序，你可以先使用 `OrderedDict` 来构建这样的数据：

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

讨论

`OrderedDict` 内部维护着一个根据键插入顺序排序的双向链表。每次当一个新的元素插入进来的时候，它会被放到链表的尾部。对于一个已经存在的键的重复赋值不会改变键的顺序。

需要注意的是，一个 `OrderedDict` 的大小是一个普通字典的两倍，因为它内部维护着另外一个链表。所以如果你要构建一个需要大量 `OrderedDict` 实例的数据结构的时候(比如读取100,000行CSV数据到一个 `OrderedDict` 列表中去)，那么你就得仔细权衡一下是否使用 `OrderedDict` 带来的好处要大过额外内存消耗的影响。

1.8 字典的运算

问题

怎样在数据字典中执行一些计算操作(比如求最小值、最大值、排序等等)?

解决方案

考虑下面的股票名和价格映射字典：

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

为了对字典值执行计算操作，通常需要使用 `zip()` 函数先将键和值反转过来。比如，下面是查找最小和最大股票价格和股票值的代码：

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')
max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')
```

类似的，可以使用 `zip()` 和 `sorted()` 函数来排列字典数据：

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),
#                  (45.23, 'ACME'), (205.55, 'IBM'),
#                  (612.78, 'AAPL')]
```

执行这些计算的时候，需要注意的是 `zip()` 函数创建的是一个只能访问一次的迭代器。比如，下面的代码就会产生错误：

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names)) # OK
print(max(prices_and_names)) # ValueError: max() arg is an empty sequence
```

讨论

如果你在一个字典上执行普通的数学运算，你会发现它们仅仅作用于键，而不是值。比如：

```
min(prices) # Returns 'AAPL'  
max(prices) # Returns 'IBM'
```

这个结果并不是你想要的，因为你想要在字典的值集合上执行这些计算。或许你会尝试着使用字典的 `values()` 方法来解决这个问题：

```
min(prices.values()) # Returns 10.75  
max(prices.values()) # Returns 612.78
```

不幸的是，通常这个结果同样也不是你想要的。你可能还想要知道对应的键的信息(比如那种股票价格是最低的？)。

你可以在 `min()` 和 `max()` 函数中提供 `key` 函数参数来获取最小值或最大值对应的键的信息。比如：

```
min(prices, key=lambda k: prices[k]) # Returns 'FB'  
max(prices, key=lambda k: prices[k]) # Returns 'AAPL'
```

但是，如果还想要得到最小值，你又得执行一次查找操作。比如：

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

前面的 `zip()` 函数方案通过将字典“反转”为(值，键)元组序列来解决上述问题。当比较两个元组的时候，值会先进行比较，然后才是键。这样的话你就能通过一条简单的语句就能很轻松的实现在字典上的求最值和排序操作了。

需要注意的是在计算操作中使用到了(值，键)对。当多个实体拥有相同的值的时候，键会决定返回结果。比如，在执行 `min()` 和 `max()` 操作的时候，如果恰巧最小或最大值有重复的，那么拥有最小或最大键的实体会返回：

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }  
>>> min(zip(prices.values(), prices.keys()))  
(45.23, 'AAA')  
>>> max(zip(prices.values(), prices.keys()))  
(45.23, 'ZZZ')  
>>>
```

1.9 查找两字典的相同点

问题

怎样在两个字典中寻寻找相同点(比如相同的键、相同的值等等)?

解决方案

考虑下面两个字典:

```
a = {
    'x' : 1,
    'y' : 2,
    'z' : 3
}

b = {
    'w' : 10,
    'x' : 11,
    'y' : 2
}
```

为了寻找两个字典的相同点, 可以简单的在两字典的 `keys()` 或者 `items()` 方法返回结果上执行集合操作。比如:

```
# Find keys in common
a.keys() & b.keys() # { 'x', 'y' }
# Find keys in a that are not in b
a.keys() - b.keys() # { 'z' }
# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }
```

这些操作也可以用于修改或者过滤字典元素。比如, 假如你想以现有字典构造一个排除几个指定键的新字典。下面利用字典推导来实现这样的需求:

```
# Make a new dictionary with certain keys removed
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

讨论

一个字典就是一个键集合与值集合的映射关系。字典的 `keys()` 方法返回一个展现键集合的键视图对象。键视图的一个很少被了解的特性就是它们也支持集合操作，比如集合并、交、差运算。所以，如果你想对集合的键执行一些普通的集合操作，可以直接使用键视图对象而不用先将它们转换成一个 `set`。

字典的 `items()` 方法返回一个包含(键, 值)对的元素视图对象。这个对象同样也支持集合操作，并且可以被用来查找两个字典有哪些相同的键值对。

尽管字典的 `values()` 方法也是类似，但是它并不支持这里介绍的集合操作。某种程度上是因为值视图不能保证所有的值互不相同，这样会导致某些集合操作会出现问题。不过，如果你硬要在值上面执行这些集合操作的话，你可以先将值集合转换成 `set`，然后再执行集合运算就行了。

1.10 删除序列相同元素并保持顺序

问题

怎样在一个序列上面保持元素顺序的同时消除重复的值？

解决方案

如果序列上的值都是 `hashable` 类型，那么可以很简单的利用集合或者生成器来解决这个问题。比如：

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

下面是使用上述函数的例子：

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

这个方法仅仅在序列中元素为 `hashable` 的时候才管用。如果你想消除元素不可哈希(比如 `dict` 类型)的序列中重复元素的话,你需要将上述代码稍微改变一下,就像这样:

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

这里的`key`参数指定了一个函数,将序列元素转换成 `hashable` 类型。下面是它的用法示例:

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

如果你想基于单个字段、属性或者某个更大的数据结构来消除重复元素,第二种方案同样可以胜任。

讨论

如果你仅仅就是想消除重复元素,通常可以简单的构造一个集合。比如:

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

然而,这种方法不能维护元素的顺序,生成的结果中的元素位置被打乱。而上面的方法可以避免这种情况。

在本节中我们使用了生成器函数让我们的函数更加通用,不仅仅是局限于列表处理。比如,如果你如果你想读取一个文件,消除重复行,你可以很容易像这样做:

```
with open(somefile, 'r') as f:
    for line in dedupe(f):
        ...
```

上述key函数参数模仿了 `sorted()`，`min()` 和 `max()` 等内置函数的相似功能。可以参考1.8和1.13小节了解更多。

1.11 命名切片

问题

你的程序已经出现一大堆已无法直视的硬编码切片下标，然后你想清理下代码。

解决方案

假定你有一段代码要从一个记录字符串中几个固定位置提取出特定的数据字段(比如文件或类似格式):

```
##### 0123456789012345678901234567890123456789012345678901234567890'
record = '.....100 .....513.25 .....'
cost = int(record[20:23]) * float(record[31:37])
```

与其那样写，为什么不想这样命名切片呢：

```
SHARES = slice(20, 23)
PRICE = slice(31, 37)
cost = int(record[SHARES]) * float(record[PRICE])
```

第二种版本中，你避免了大量无法理解的硬编码下标，使得你的代码更加清晰可读了。

讨论

一般来讲，代码中如果出现大量的硬编码下标值会使得可读性和可维护性大大降低。比如，如果你回过来看看一年前你写的代码，你会摸着脑袋想那时候自己到底想干嘛啊。这里的解决方案是一个很简单的方法让你更加清晰的表达代码到底要做什么。

内置的 `slice()` 函数创建了一个切片对象，可以被用在任何切片允许使用的地方。比如：

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

如果你有一个切片对象 `s`，你可以分别调用它的 `s.start`，`s.stop`，`s.step` 属性来获取更多的信息。比如：

```
>>> s = slice(5, 50, 2)
>>> s.start
5
>>> s.stop
50
>>> s.step
2
>>>
```

另外，你还能通过调用切片的 `indices(size)` 方法将它映射到一个确定大小的序列上，这个方法返回一个三元组 `(start, stop, step)`，所有值都会被合适的缩小以满足边界限制，从而使用的时候避免出现 `IndexError` 异常。比如：

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
w
r
d
>>>
```

1.12 序列中出现次数最多的元素

问题

怎样找出一个序列中出现次数最多的元素呢？

解决方案

`collections.Counter` 类就是专门为这类问题而设计的，它甚至有一个有用的 `most_common()` 方法直接给了你答案。

为了演示，先假设你有一个单词列表并且想找出哪个单词出现频率最高。你可以这样做：

```
words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
    'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
    'my', 'eyes', "you're", 'under'
]
from collections import Counter
word_counts = Counter(words)
# 出现频率最高的3个单词
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

讨论

作为输入，`Counter` 对象可以接受任意的 `hashable` 序列对象。在底层实现上，一个 `Counter` 对象就是一个字典，将元素映射到它出现的次数上。比如：

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

如果你想手动增加计数，可以简单的用加法：

```
>>> morewords = ['why', 'are', 'you', 'not', 'looking', 'in', 'my', 'eyes']
>>> for word in morewords:
...     word_counts[word] += 1
...
>>> word_counts['eyes']
9
>>>
```

或者你可以使用 `update()` 方法:

```
>>> word_counts.update(morewords)
>>>
```

`Counter` 实例一个鲜为人知的特性是它们可以很容易的跟数学运算操作相结合。比如:

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
'you're': 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
'my': 1, 'why': 1})
>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
'looking': 1, 'are': 1, 'under': 1, 'you': 1})
>>> # Subtract counts
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
'you're': 1, "don't": 1, 'under': 1})
>>>
```

毫无疑问, `Counter` 对象在几乎所有需要制表或者计数数据的场合是非常有用的工具。在解决这类问题的时候你应该优先选择它, 而不是手动的利用字典去实现。

1.13 通过某个关键字排序一个字典列表

问题

你有一个字典列表, 你想根据某个或某几个字典字段来排序这个列表。

解决方案

通过使用 `operator` 模块的 `itemgetter` 函数，可以非常容易的排序这样的数据结构。假设你从数据库中检索出来网站会员信息列表，并且以下列的数据结构返回：

```
rows = [  
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},  
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},  
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},  
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}  
]
```

根据任意的字典字段来排序输入结果行是很容易实现的，代码示例：

```
from operator import itemgetter  
rows_by_fname = sorted(rows, key=itemgetter('fname'))  
rows_by_uid = sorted(rows, key=itemgetter('uid'))  
print(rows_by_fname)  
print(rows_by_uid)
```

代码的输出如下：

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},  
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},  
{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},  
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]  
[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},  
{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},  
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},  
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

`itemgetter()` 函数也支持多个keys，比如下面的代码

```
rows_by_lfname = sorted(rows, key=itemgetter('lname', 'fname'))  
print(rows_by_lfname)
```

会产生如下的输出：

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},  
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},  
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},  
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

讨论

在上面例子中，`rows` 被传递给接受一个关键字参数的 `sorted()` 内置函数。这个参数是 `callable` 类型，并且从 `rows` 中接受一个单一元素，然后返回被用来排序的值。

`itemgetter()` 函数就是负责创建这个 `callable` 对象的。

`operator.itemgetter()` 函数有一个被 `rows` 中的记录用来查找值的索引参数。可以是一个字典键名称，一个整形值或者任何能够传入一个对象的 `__getitem__()` 方法的值。如果你传入多个索引参数给 `itemgetter()`，它生成的 `callable` 对象会返回一个包含所有元素值的元组，并且 `sorted()` 函数会根据这个元组中元素顺序去排序。但你想要同时在几个字段上面进行排序(比如通过姓和名来排序，也就是例子中的那样)的时候这种方法是很有用的。

`itemgetter()` 有时候也可以用 `lambda` 表达式代替，比如：

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])  
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'], r['fname']))
```

这种方案也不错。但是，使用 `itemgetter()` 方式会运行的稍微快点。因此，如果你对性能要求比较高的话就使用 `itemgetter()` 方式。

最后，不要忘了这节中展示的技术也同样适用于 `min()` 和 `max()` 等函数。比如：

```
>>> min(rows, key=itemgetter('uid'))  
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}  
>>> max(rows, key=itemgetter('uid'))  
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}  
>>>
```

1.14 排序不支持原生比较的对象

问题

你想排序类型相同的对象，但是他们不支持原生的比较操作。

解决方案

内置的 `sorted()` 函数有一个关键字参数 `key`，可以传入一个 `callable` 对象给它，这个 `callable` 对象对每个传入的对象返回一个值，这个值会被 `sorted` 用来排序这些对象。比如，如果你在应用程序里面有一个 `User` 实例序列，并且你希望通过他们的 `user_id` 属性进行排序，你可以提供一个以 `User` 实例作为输入并输出对应 `user_id` 值的 `callable` 对象。比如：

```
class User:
    def __init__(self, user_id):
        self.user_id = user_id

    def __repr__(self):
        return 'User({})'.format(self.user_id)

def sort_notcompare():
    users = [User(23), User(3), User(99)]
    print(users)
    print(sorted(users, key=lambda u: u.user_id))
```

另外一种方式是使用 `operator.attrgetter()` 来代替 `lambda` 函数：

```
>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>
```

讨论

选择使用 `lambda` 函数或者是 `attrgetter()` 可能取决于个人喜好。但是，`attrgetter()` 函数通常会运行的快点，并且还能同时允许多个字段进行比较。这个跟 `operator.itemgetter()` 函数作用于字典类型很类似(参考1.13小节)。例如，如果 `User` 实例还有一个 `first_name` 和 `last_name` 属性，那么可以向下面这样排序：

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

同样需要注意的是，这一小节用到的技术同样适用于像 `min()` 和 `max()` 之类的函数。比如：

```
>>> min(users, key=attrgetter('user_id'))
User(3)
>>> max(users, key=attrgetter('user_id'))
User(99)
>>>
```

1.15 通过某个字段将记录分组

问题

你有一个字典或者实例的序列，然后你想根据某个特定的字段比如 `date` 来分组迭代访问。

解决方案

`itertools.groupby()` 函数对于这样的数据分组操作非常实用。为了演示，假设你已经有了下列的字典列表：

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

现在假设你想在按 `date` 分组后的数据块上进行迭代。为了这样做，你首先需要按照指定的字段(这里就是 `date`)排序，然后调用 `itertools.groupby()` 函数：

```
from operator import itemgetter
from itertools import groupby

# Sort by the desired field first
rows.sort(key=itemgetter('date'))
# Iterate in groups
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print(' ', i)
```

运行结果:

```
07/01/2012
 {'date': '07/01/2012', 'address': '5412 N CLARK'}
 {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
 {'date': '07/02/2012', 'address': '5800 E 58TH'}
 {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
 {'date': '07/02/2012', 'address': '1060 W ADDISON'}
07/03/2012
 {'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
 {'date': '07/04/2012', 'address': '5148 N CLARK'}
 {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

讨论

`groupby()` 函数扫描整个序列并且查找连续相同值(或者根据指定key函数返回值相同)的元素序列。在每次迭代的时候,它会返回一个值和一个迭代器对象,这个迭代器对象可以生成元素值全部等于上面那个值的组中所有对象。

一个非常重要的准备步骤是要根据指定的字段将数据排序。因为 `groupby()` 仅仅检查连续的元素,如果事先并没有排序完成的话,分组函数将得不到想要的结果。

如果你仅仅只是想根据date字段将数据分组到一个大的数据结构中去,并且允许随机访问,那么你最好使用 `defaultdict()` 来构建一个多值字典,关于多值字典已经在1.6小节有过详细的介绍。比如:

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```


这样的话你可以很轻松的就能对每个指定日期访问对应的记录：

```
>>> for r in rows_by_date['07/01/2012']:
...     print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

在上面这个例子中，我们没有必要先将记录排序。因此，如果对内存占用不是很关心，这种方式会比先排序然后再通过 `groupby()` 函数迭代的方式运行得快一些。

1.16 过滤序列元素

问题

你有一个数据序列，想利用一些规则从中提取出需要的值或者是缩短序列

解决方案

最简单的过滤序列元素的方法就是使用列表推导。比如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

使用列表推导的一个潜在缺陷就是如果输入非常大的时候会产生一个非常大的结果集，占用大量内存。如果你对内存比较敏感，那么你可以使用生成器表达式迭代产生过滤的元素。比如：

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
...     print(x)
...
1
4
10
2
3
>>>
```

有时候，过滤规则比较复杂，不能简单的在列表推导或者生成器表达式中表达出来。比如，假设过滤的时候需要处理一些异常或者其他复杂情况。这时候你可以将过滤代码放到一个函数中，然后使用内建的 `filter()` 函数。示例如下：

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']
def is_int(val):
    try:
        x = int(val)
        return True
    except ValueError:
        return False
ivals = list(filter(is_int, values))
print(ivals)
# Outputs ['1', '2', '-3', '4', '5']
```

`filter()` 函数创建了一个迭代器，因此如果你想得到一个列表的话，就得像示例那样使用 `list()` 去转换。

讨论

列表推导和生成器表达式通常情况下是过滤数据最简单的方式。其实它们还能在过滤的时候转换数据。比如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>
```

过滤操作的一个变种就是将不符合条件的值用新的值代替，而不是丢弃它们。比如，在一列数据中你可能不仅想找到正数，而且还想将不是正数的数替换成指定的数。通过将过滤条件放到条件表达式中去，可以很容易的解决这个问题，就像这样：

```
>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

另外一个值得关注的过滤工具就是 `itertools.compress()`，它以一个 `iterable` 对象和一个相对应的 `Boolean` 选择器序列作为输入参数。然后输出 `iterable` 对象中对应选择器为 `True` 的元素。当你需要用另外一个相关联的序列来过滤某个序列的时候，这个函数是非常有用的。比如，假如现在你有下面两列数据：

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK',
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]
counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

现在你想将那些对应 `count` 值大于5的地址全部输出，那么你可以这样做：

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']
>>>
```

这里的关键点在于先创建一个 `Boolean` 序列，指示哪些元素复合条件。然后 `compress()` 函数根据这个序列去选择输出对应位置为 `True` 的元素。

和 `filter()` 函数类似, `compress()` 也是返回的一个迭代器。因此, 如果你需要得到一个列表, 那么你需要使用 `list()` 来将结果转换为列表类型。

1.17 从字典中提取子集

问题

你想构造一个字典, 它是另外一个字典的子集。

解决方案

最简单的方式是使用字典推导。比如:

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
# Make a dictionary of all prices over 200
p1 = {key: value for key, value in prices.items() if value > 200}
# Make a dictionary of tech stocks
tech_names = {'AAPL', 'IBM', 'HPQ', 'MSFT'}
p2 = {key: value for key, value in prices.items() if key in tech_names}
```

讨论

大多数情况下字典推导能做到的, 通过创建一个元组序列然后把它传给 `dict()` 函数也能实现。比如:

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

但是, 字典推导方式表意更清晰, 并且实际上也会运行的更快些 (在这个例子中, 实际测试几乎比 `dict()` 函数方式快整整一倍)。

有时候完成同一件事会有多种方式。比如, 第二个例子程序也可以像这样重写:

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

但是，运行时间测试结果显示这种方案大概比第一种方案慢1.6倍。如果对程序运行性能要求比较高的话，需要花点时间去做计时测试。关于更多计时和性能测试，可以参考14.13小节

1.18 映射名称到序列元素

问题

你有一段通过下标访问列表或者元组中元素的代码，但是这样有时候会使得你的代码难以阅读，于是你想通过名称来访问元素。

解决方案

`collections.namedtuple()` 函数通过使用一个普通的元组对象来帮你解决这个问题。这个函数实际上是一个返回Python中标准元组类型子类的一个工厂方法。你需要传递一个类型名和你需要的字段给它，然后它就会返回一个类，你可以初始化这个类，为你定义的字段传递值等。代码示例：

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

尽管 `namedtuple` 的实例看起来像一个普通的类实例，但是它跟元组类型是可交换的，支持所有的普通元组操作，比如索引和解压。比如：

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

命名元组的一个主要用途是将你的代码从下标操作中解脱出来。因此，如果你从数据库调用中返回了一个很大的元组列表，通过下标去操作其中的元素，当你在表中添加了新的列的时候你的代码可能就会出错了。但是如果你使用了命名元组，那么就不会有这样的顾虑。

为了说明清楚，下面是使用普通元组的代码：

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

下标操作通常会让代码表意不清晰，并且非常依赖记录的结构。下面是使用命名元组的版本：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

讨论

命名元组另一个用途就是作为字典的替代，因为字典存储需要更多的内存空间。如果你需要构建一个非常大的包含字典的数据结构，那么使用命名元组会更加高效。但是需要注意的是，不像字典那样，一个命名元组是不可更改的。比如：

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

如果你真的需要改变然后的属性，那么可以使用命名元组实例的 `_replace()` 方法，它会创建一个全新的命名元组并将对应的字段用新的值取代。比如：

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

`_replace()` 方法还有一个很有用的特性就是当你的命名元组拥有可选或者缺失字段时候，它是一个非常方便的填充数据的方法。你可以先创建一个包含缺省值的原型元组，然后使用 `_replace()` 方法创建新的值被更新过的实例。比如：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Create a prototype instance
stock_prototype = Stock('', 0, 0.0, None, None)

# Function to convert a dictionary to a Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

下面是它的使用方法：

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

最后要说的是，如果你的目标是定义一个需要更新很多实例属性的高效数据结构，那么命

名元组并不是你的最佳选择。这时候你应该考虑定义一个包含 `__slots__` 方法的类(参考 8.4 小节)。

1.19 转换并同时计算数据

问题

你需要在数据序列上执行聚集函数(比如 `sum()`, `min()`, `max()`), 但是首先你需要先转换或者过滤数据

解决方案

一个非常优雅的方式去结合数据计算与转换就是使用一个生成器表达式参数。比如, 如果你想计算平方和, 可以像下面这样做:

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

下面是更多的例子:

```
# Determine if any .py files exist in a directory
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')
# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))
# Data reduction across fields of a data structure
portfolio = [
    {'name': 'GOOG', 'shares': 50},
    {'name': 'YHOO', 'shares': 75},
    {'name': 'AOL', 'shares': 20},
    {'name': 'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)
```

讨论

上面的示例向你演示了当生成器表达式作为一个单独参数传递给函数时候的巧妙语法(你并不需要多加一个括号)。比如, 下面这些语句是等效的:

```
s = sum((x * x for x in nums)) # 显示的传递一个生成器表达式对象
s = sum(x * x for x in nums) # 更加优雅的实现方式, 省略了括号
```

使用一个生成器表达式作为参数会比先创建一个临时列表更加高效和优雅。比如, 如果你不使用生成器表达式的话, 你可能会考虑使用下面的实现方式:

```
nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])
```

这种方式同样可以达到想要的效果, 但是它会多一个步骤, 先创建一个额外的列表。对于小型列表可能没什么关系, 但是如果元素数量非常大的时候, 它会创建一个巨大的仅仅被使用一次就被丢弃的临时数据结构。而生成器方案会以迭代的方式转换数据, 因此更省内存。

在使用一些聚集函数比如 `min()` 和 `max()` 的时候你可能更加倾向于使用生成器版本, 它们接受的一个key关键字参数或许对你很有帮助。比如, 在上面的证券例子中, 你可能会考虑下面的实现版本:

```
# Original: Returns 20
min_shares = min(s['shares'] for s in portfolio)
# Alternative: Returns {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])
```

1.20 合并多个字典或映射

问题

现在有多个字典或者映射, 你想将它们从逻辑上合并为一个单一的映射后执行某些操作, 比如查找值或者检查某些键是否存在。

解决方案

加入你有如下两个字典:

```
a = {'x': 1, 'z': 3 }
b = {'y': 2, 'z': 4 }
```

现在假设你必须在两个字典中执行查找操作(比如先从 `a` 中找, 如果找不到再在 `b` 中找)。一个非常简单扼解决方案就是使用 `collections` 模块中的 `ChainMap` 类。比如:

```
from collections import ChainMap
c = ChainMap(a,b)
print(c['x']) # Outputs 1 (from a)
print(c['y']) # Outputs 2 (from b)
print(c['z']) # Outputs 3 (from a)
```

讨论

一个 `ChainMap` 接受多个字典并将它们在逻辑上变为一个字典。然后, 这些字典并不是真的合并在一起了, `ChainMap` 类只是在内部创建了一个容纳这些字典的列表并重新定义了一些常见的字典操作来遍历这个列表。大部分字典操作都是可以正常使用的, 比如:

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

如果出现重复键, 那么第一次出现的映射值会被返回。因此, 例子程序中的 `c['z']` 总是会返回字典 `a` 中对应的值, 而不是 `b` 中对应的值。

对于字典的更新或删除操作总是影响的是列表中第一个字典。比如:

```

>>> c['z'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
KeyError: "Key not found in the first mapping: 'y'"
>>>

```

`ChainMap` 对于编程语言中的作用范围变量(比如 `globals` , `locals` 等)是非常有用的。事实上, 有一些方法可以使它变得简单:

```

>>> values = ChainMap()
>>> values['x'] = 1
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 2
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 3
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
2
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
1
>>> values
ChainMap({'x': 1})
>>>

```

作为 `ChainMap` 的替代, 你可能会考虑使用 `update()` 方法将两个字典合并。比如:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

这样也能行得通，但是它需要你创建一个完全不同的字典对象(或者是破坏现有字典结构)。同时，如果原字典做了更新，这种改变不会反应到新的合并字典中去。比如：

```
>>> a['x'] = 13
>>> merged['x']
1
```

ChainMap 使用原来的字典，它自己不创建新的字典。所以它并不会产生上面所说的结果，比如：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x'] # Notice change to merged dicts
42
>>>
```

第二章：字符串和文本

几乎所有有用的程序都会涉及到某些文本处理，不管是解析数据还是产生输出。这一章将重点关注文本的操作处理，比如提取字符串，搜索，替换以及解析等。大部分的问题都能简单的调用字符串的内建方法完成。但是，一些更为复杂的操作可能需要正则表达式或者强大的解析器，所有这些主题我们都会详细讲解。并且在操作Unicode时候碰到的一些棘手的问题在这里也会被提及到。

Contents:

2.1 使用多个界定符分割字符串

问题

你需要将一个字符串分割为多个字段，但是分隔符(还有周围的空格)并不是固定的。

解决方案

`string` 对象的 `split()` 方法只适应于非常简单的字符串分割情形，它并不允许有多个分隔符或者是分隔符周围不确定的空格。当你需要更加灵活的切割字符串的时候，最好使用 `re.split()` 方法：

```
>>> line = 'asdf fjdk; afed, fjek,asdf, foo'
>>> import re
>>> re.split(r'[;,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

讨论

函数 `re.split()` 是非常实用的，因为它允许你为分隔符指定多个正则模式。比如，在上面的例子中，分隔符可以是逗号，分号或者是空格，并且后面紧跟着任意个的空格。只要这个模式被找到，那么匹配的分隔符两边的实体都会被当成是结果中的元素返回。返回结果为一个字段列表，这个跟 `str.split()` 返回值类型是一样的。

当你使用 `re.split()` 函数时候，需要特别注意的是正则表达式中是否包含一个括号捕获分组。如果使用了捕获分组，那么被匹配的文本也将出现在结果列表中。比如，观察一下这段代码运行后的结果：

```
>>> fields = re.split(r'(;|,|\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf', ',', 'foo']
>>>
```

获取分割字符在某些情况下也是有用的。比如，你可能想保留分割字符串，用来在后面重新构造一个新的输出字符串：

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ', ', ', ', ', ', '']
>>> # Reform the line using the same delimiters
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

如果你不想保留分割字符串到结果列表中去，但仍然需要使用到括号来分组正则表达式的话，确保你的分组是非捕获分组，形如 `(?:...)`。比如：

```
>>> re.split(r'(?:,|;|\s)\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>>
```

2.2 字符串开头或结尾匹配

问题

你需要通过指定的文本模式去检查字符串的开头或者结尾，比如文件名后缀，URL Scheme等等。

解决方案

检查字符串开头或结尾的一个简单方法是使用 `str.startswith()` 或者是 `str.endswith()` 方法。比如：

```
>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>
```

如果你想检查多种匹配可能，只需要将所有的匹配项放入到一个元组中去，然后传给

`startswith()` 或者 `endswith()` 方法：

```
>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
>>> [name for name in filenames if name.endswith(('.c', '.h')) ]
['foo.c', 'spam.c', 'spam.h']
>>> any(name.endswith('.py') for name in filenames)
True
>>>
```

下面是另一个例子：

```
from urllib.request import urlopen

def read_data(name):
    if name.startswith(('http:', 'https:', 'ftp:')):
        return urlopen(name).read()
    else:
        with open(name) as f:
            return f.read()
```

奇怪的是，这个方法中必须要输入一个元组作为参数。如果你恰巧有一个 `list` 或者 `set` 类型的选择项，要确保传递参数前先调用 `tuple()` 将其转换为元组类型。比如：

```
>>> choices = ['http:', 'ftp:']
>>> url = 'http://www.python.org'
>>> url.startswith(choices)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: startswith first arg must be str or a tuple of str, not list
>>> url.startswith(tuple(choices))
True
>>>
```

讨论

`startswith()` 和 `endswith()` 方法提供了一个非常方便的方式去做字符串开头和结尾的检查。类似的操作也可以使用切片来实现，但是代码看起来没有那么优雅。比如：

```
>>> filename = 'spam.txt'
>>> filename[-4:] == '.txt'
True
>>> url = 'http://www.python.org'
>>> url[:5] == 'http:' or url[:6] == 'https:' or url[:4] == 'ftp:'
True
>>>
```

你可以能还想使用正则表达式去实现，比如：

```
>>> import re
>>> url = 'http://www.python.org'
>>> re.match('http:|https:|ftp:', url)
<_sre.SRE_Match object at 0x101253098>
>>>
```

这种方式也行得通，但是对于简单的匹配实在是有点小材大用了，本节中的方法更加简单并且运行会更快些。

最后提一下，当和其他操作比如普通数据聚合相结合的时候 `startswith()` 和 `endswith()` 方法是很不错的。比如，下面这个语句检查某个文件夹中是否存在指定的文件类型：

```
if any(name.endswith(('.c', '.h'))) for name in listdir(dirname):
    ...
```

2.3 用Shell通配符匹配字符串

问题

你想使用 **Unix Shell** 中常用的通配符(比如 `*.py` , `Dat[0-9]*.csv` 等)去匹配文本字符串

解决方案

`fnmatch` 模块提供了两个函数—— `fnmatch()` 和 `fnmatchcase()` ，可以用来实现这样的匹配。用法如下：


```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('foo.txt', '*.txt')
True
>>> fnmatch('foo.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')
True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

`fnmatch()` 函数使用底层操作系统的大小写敏感规则(不同的系统是不一样的)来匹配模式。比如:

```
>>> # On OS X (Mac)
>>> fnmatch('foo.txt', '*.TXT')
False
>>> # On Windows
>>> fnmatch('foo.txt', '*.TXT')
True
>>>
```

如果你对这个区别很在意, 可以使用 `fnmatchcase()` 来代替。它完全使用你的模式大小写匹配。比如:

```
>>> fnmatchcase('foo.txt', '*.TXT')
False
>>>
```

这两个函数通常会被忽略的一个特性是在处理非文件名的字符串时候它们也是很有用的。比如, 假设你有一个街道地址的列表数据:

```
addresses = [
    '5412 N CLARK ST',
    '1060 W ADDISON ST',
    '1039 W GRANVILLE AVE',
    '2122 N CLARK ST',
    '4802 N BROADWAY',
]
```

你可以像这样写列表推导:

```
>>> from fnmatch import fnmatchcase
>>> [addr for addr in addresses if fnmatchcase(addr, '* ST')]
['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']
>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9] *CLARK*')]
['5412 N CLARK ST']
>>>
```

讨论

`fnmatch()` 函数匹配能力介于简单的字符串方法和强大的正则表达式之间。如果在数据处理操作中只需要简单的通配符就能完成的时候，这通常是一个比较合理的方案。

如果你的代码需要做文件名的匹配，最好使用 `glob` 模块。参考5.13小节。

2.4 字符串匹配和搜索

问题

你想匹配或者搜索特定模式的文本

解决方案

如果你想匹配的是字面字符串，那么你通常只需要调用基本字符串方法就行，比如

`str.find()`，`str.endswith()`，`str.startswith()` 或者类似的方法：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> # Exact match
>>> text == 'yeah'
False
>>> # Match at start or end
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False
>>> # Search for the location of the first occurrence
>>> text.find('no')
10
>>>
```

对于复杂的匹配需要使用正则表达式和 `re` 模块。为了解释正则表达式的基本原理，假设你想匹配数字格式的日期字符串比如 `11/27/2012`，你可以这样做：

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re
>>> # Simple matching: \d+ means match one or more digits
>>> if re.match(r'\d+/\d+/\d+', text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
...     print('yes')
... else:
...     print('no')
...
no
>>>
```

如果你想使用同一个模式去做多次匹配，你应该先将模式字符串预编译为模式对象。比如：

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if datepat.match(text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if datepat.match(text2):
...     print('yes')
... else:
...     print('no')
...
no
>>>
```

`match()` 总是从字符串开始去匹配，如果你想查找字符串任意部分的模式出现位置，使用 `findall()` 方法去代替。比如：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

在定义正则式的时候，通常会利用括号去捕获分组。比如：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

捕获分组可以使得后面的处理更加简单，因为可以分别将每个组的内容提取出来。比如：

```
>>> m = datepat.match('11/27/2012')
>>> m
<_sre.SRE_Match object at 0x1005d2750>
>>> # Extract the contents of each group
>>> m.group(0)
'11/27/2012'
>>> m.group(1)
'11'
>>> m.group(2)
'27'
>>> m.group(3)
'2012'
>>> m.groups()
('11', '27', '2012')
>>> month, day, year = m.groups()
>>>
>>> # Find all matches (notice splitting into tuples)
>>> text
'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>> for month, day, year in datepat.findall(text):
...     print('{}-{}-{}'.format(year, month, day))
...
2012-11-27
2013-3-13
>>>
```

`findall()` 方法会搜索文本并以列表形式返回所有的匹配。如果你想以迭代方式返回匹配，可以使用 `finditer()` 方法来代替，比如：

```
>>> for m in datepat.finditer(text):
...     print(m.groups())
...
('11', '27', '2012')
('3', '13', '2013')
>>>
```

讨论

关于正则表达式理论的教程已经超出了本书的范围。不过，这一节阐述了使用`re`模块进行匹配和搜索文本的最基本方法。核心步骤就是先使用 `re.compile()` 编译正则表达式字符串，然后使用 `match()`，`findall()` 或者 `finditer()` 等方法。

当写正则式字符串的时候，相对普遍的做法是使用原始字符串比如 `r'(\d+)/(\d+)/(\d+)'`。这种字符串将不去解析反斜杠，这在正则表达式中是很有用的。如果不这样做的话，你必须使用两个反斜杠，类似 `'(\\d+)/\\d+)/\\d+)'`。

需要注意的是 `match()` 方法仅仅检查字符串的开始部分。它的匹配结果有可能并不是你期望的那样。比如：

```
>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2012'
>>>
```

如果你想精确匹配，确保你的正则表达式以`$`结尾，就像这么这样：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

最后，如果你仅仅是做一次简单的文本匹配/搜索操作的话，可以略过编译部分，直接使用 `re` 模块级别的函数。比如：

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

但是需要注意的是，如果你打算做大量的匹配和搜索操作的话，最好先编译正则表达式，然后再重复使用它。模块级别的函数会将最近编译过的模式缓存起来，因此并不会消耗太多的性能，但是如果使用预编译模式的话，你将会减少查找和一些额外的处理损耗。

2.5 字符串搜索和替换

问题

你想在字符串中搜索和匹配指定的文本模式

解决方案

对于简单的字面模式，直接使用 `str.replace()` 方法即可，比如：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

对于复杂的模式，请使用 `re` 模块中的 `sub()` 函数。为了说明这个，假设你想将形式为 `11/27/201` 的日期字符串改成 `2012-11-27`。示例如下：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

`sub()` 函数中的第一个参数是被匹配的模式，第二个参数是替换模式。反斜杠数字比如 `\3` 指向前面模式的捕获组号。

如果你打算用相同的模式做多次替换，考虑先编译它来提升性能。比如：

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

对于更加复杂的替换，可以传递一个替换回调函数来代替，比如：

```
>>> from calendar import month_abbr
>>> def change_date(m):
...     mon_name = month_abbr[int(m.group(1))]
...     return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

一个替换回调函数的参数是一个 `match` 对象，也就是 `match()` 或者 `find()` 返回的对象。使用 `group()` 方法来提取特定的匹配部分。回调函数最后返回替换字符串。

如果除了替换后的结果外，你还想知道有多少替换发生了，可以使用 `re.subn()` 来代替。比如：

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>> n
2
>>>
```

讨论

关于正则表达式搜索和替换，上面演示的 `sub()` 方法基本已经涵盖了所有。其实最难的部分就是编写正则表达式模式，这个最好是留给作者自己去练习了。

2.6 字符串忽略大小写的搜索替换

问题

你需要以忽略大小写的方式搜索与替换文本字符串

解决方案

为了在文本操作时忽略大小写，你需要在使用 `re` 模块的时候给这些操作提供 `re.IGNORECASE` 标志参数。比如：

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

最后的那个例子揭示了一个小缺陷，替换字符串并不会自动跟被匹配字符串的大小写保持一致。为了修复这个，你可能需要一个辅助函数，就像下面的这样：

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
            return word.upper()
        elif text.islower():
            return word.lower()
        elif text[0].isupper():
            return word.capitalize()
        else:
            return word
    return replace
```

下面是使用上述函数的方法：

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECASE)
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

译者注：`matchcase('snake')` 返回了一个回调函数(参数必须是 `match` 对象)，前面一节一节提到过，`sub()` 函数除了接受替换字符串外，还能接受一个回调函数。

讨论

对于一般的忽略大小写的匹配操作，简单的传递一个 `re.IGNORECASE` 标志参数就已经足够了。但是需要注意的是，这个对于某些需要大小写转换的Unicode匹配可能还不够，参考2.10小节了解更多细节。

2.7 最短匹配模式

问题

你正在试着用正则表达式匹配某个文本模式，但是它找到的是模式的最长可能匹配。而你想修改它变成查找最短的可能匹配。

解决方案

这个问题一般出现在需要匹配一对分隔符之间的文本的时候(比如引号包含的字符串)。为了说明清楚，考虑如下的例子：

```
>>> str_pat = re.compile(r'\"(.*?)\"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no." Phone says "yes.']
>>>
```

在这个例子中，模式 `r'\"(.*?)\"'` 的意图是匹配被双引号包含的文本。但是在正则表达式中`*`操作符是贪婪的，因此匹配操作会查找最长的可能匹配。于是在第二个例子中搜索 `text2` 的时候返回结果并不是我们想要的。

为了修正这个问题，可以在模式中的`*`操作符后面加上`?`修饰符，就像这样：

```
>>> str_pat = re.compile(r'\"(.*?)\"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

这样就使得匹配变成非贪婪模式，从而得到最短的匹配，也就是我们想要的结果。

讨论

这一节展示了在写包含点(.)字符的正则表达式的时候遇到的一些常见问题。在一个模式字符串中，点(.)匹配除了换行外的任何字符。然而，如果你将点(.)号放在开始与结束符(比如引号)之间的时候，那么匹配操作会查找符合模式的最长可能匹配。这样通常会导致很多中间的被开始与结束符包含的文本被忽略掉，并最终被包含在匹配结果字符串中返回。通过在 `*` 或者 `+` 这样的操作符后面添加一个 `?` 可以强制匹配算法改成寻找最短的可能匹配。

2.8 多行匹配模式

问题

你正在试着使用正则表达式去匹配一大块的文本，而你需要跨越多行去匹配。

解决方案

这个问题很典型的出现在当你用点(.)去匹配任意字符的时候，忘记了点(.)不能匹配换行符的事实。比如，假设你想试着去匹配C语言分割的注释：

```
>>> comment = re.compile(r'/*(.*?)\*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
... multiline comment */
... '''
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

为了修正这个问题，你可以修改模式字符串，增加对换行的支持。比如：

```
>>> comment = re.compile(r'/*((?:.|\\n)*?)\*/')
>>> comment.findall(text2)
[' this is a\\n multiline comment ']
>>>
```

在这个模式中，`(?:.|\\n)` 指定了一个非捕获组(也就是它定义了一个仅仅用来做匹配，而不能通过单独捕获或者编号的组)。

讨论

`re.compile()` 函数接受一个标志参数叫 `re.DOTALL`，在这里非常有用。它可以让正则表达式中的点(.)匹配包括换行符在内的任意字符。比如：

```
>>> comment = re.compile(r'/*(.*?)\*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\n multiline comment ']
```

对于简单的情况使用 `re.DOTALL` 标记参数工作的很好，但是如果模式非常复杂或者是为了构造字符串令牌而将多个模式合并起来(2.18节有详细描述)，这时候使用这个标记参数就可能出现一些问题。如果你选择的话，最好还是定义自己的正则表达式模式，这样它可以在不需要额外的标记参数下也能工作的很好。

2.9 将Unicode文本标准化

问题

你正在处理Unicode字符串，需要确保所有字符串在底层有相同的表示。

解决方案

在Unicode中，某些字符能够用多个合法的编码表示。为了说明，考虑下面的这个例子：

```
>>> s1 = 'Spicy Jalape\u00f1o'
>>> s2 = 'Spicy Jalapen\u0303o'
>>> s1
'Spicy Jalape\u00f1o'
>>> s2
'Spicy Jalape\u00f1o'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
15
>>>
```

这里的文本“Spicy Jalape\u00f1o”使用了两种形式来表示。第一种使用整体字符“\u00f1”(U+00F1)，第二种使用拉丁字母“n”后面跟一个“~”的组合字符(U+0303)。

在需要比较字符串的程序中使用字符的多种表示会产生问题。为了修正这个问题，你可以使用`unicodedata`模块先将文本标准化：

```

>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\xf1o'
>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>

```

`normalize()` 第一个参数指定字符串标准化的方式。`NFC`表示字符应该是整体组成(比如可能的话就使用单一编码)，而`NFD`表示字符应该分解为多个组合字符表示。

Python同样支持扩展的标准化形式`NFKC`和`NFKD`，它们在处理某些字符的时候增加了额外的兼容特性。比如：

```

>>> s = '\ufb01' # A single character
>>> s
'fi'
>>> unicodedata.normalize('NFD', s)
'fi'
# Notice how the combined letters are broken apart here
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>

```

讨论

标准化对于任何需要以一致的方式处理Unicode文本的程序都是非常重要的。当处理来自用户输入的字符串而你很难去控制编码的时候尤其如此。

在清理和过滤文本的时候字符的标准化也是很重要的。比如，假设你想清除掉一些文本上面的变音符的时候(可能是为了搜索和匹配)：

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for c in t1 if not unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

最后一个例子展示了 `unicodedata` 模块的另一个重要方面，也就是测试字符类的工具函数。 `combining()` 函数可以测试一个字符是否为和音字符。在这个模块中还有其他函数用于查找字符类别，测试是否为数字字符等等。

Unicode显然是一个很大的主题。如果想更深入的了解关于标准化方面的信息，请看考 [Unicode官网中关于这部分的说明](#) Ned Batchelder在 [他的网站](#) 上对Python的Unicode处理问题也有一个很好的介绍。

2.10 在正则式中使用Unicode

问题

你正在使用正则表达式处理文本，但是关注的是Unicode字符处理。

解决方案

默认情况下 `re` 模块已经对一些Unicode字符类有了基本的支持。比如， `\\d` 已经匹配任意的unicode数字字符了：

```
>>> import re
>>> num = re.compile('\\d+')
>>> # ASCII digits
>>> num.match('123')
<_sre.SRE_Match object at 0x1007d9ed0>
>>> # Arabic digits
>>> num.match('\\u0661\\u0662\\u0663')
<_sre.SRE_Match object at 0x101234030>
>>>
```

如果你想在模式中包含指定的Unicode字符，你可以使用Unicode字符对应的转义序列(比如 `\\uFFF` 或者 `\\UFFFFFF`)。比如，下面是一个匹配几个不同阿拉伯编码页面中所有字符的正则表达式：

```
>>> arabic = re.compile('[\\u0600-\\u06ff\\u0750-\\u077f\\u08a0-\\u08ff]+')
>>>
```

当执行匹配和搜索操作的时候，最好是先标准化并且清理所有文本为标准化格式(参考2.9小节)。但是同样也应该注意一些特殊情况，比如在忽略大小写匹配和大小写转换时的行为。

```
>>> pat = re.compile('stra\u00dfe', re.IGNORECASE)
>>> s = 'straße'
>>> pat.match(s) # Matches
<_sre.SRE_Match object at 0x10069d370>
>>> pat.match(s.upper()) # Doesn't match
>>> s.upper() # Case folds
'STRASSE'
>>>
```

讨论

混合使用Unicode和正则表达式通常会让你抓狂。如果你真的打算这样做的话，最好考虑下安装第三方正则式库，它们会为Unicode的大小写转换和其他大量有趣特性提供全面的支持，包括模糊匹配。

2.11 删除字符串中不需要的字符

问题

你想去掉文本字符串开头，结尾或者中间不想要的字符，比如空白。

解决方案

`strip()` 方法能用于删除开始或结尾的字符。`rstrip()` 和 `lstrip()` 分别从左和从右执行删除操作。默认情况下，这些方法会去除空白字符，但是你也可以指定其他字符。比如：

```
>>> # Whitespace stripping
>>> s = ' hello world \n'
>>> s.strip()
'hello world'
>>> s.lstrip()
'hello world \n'
>>> s.rstrip()
' hello world'
>>>
>>> # Character stripping
>>> t = '-----hello====='
>>> t.lstrip('-')
'hello====='
>>> t.strip('-=')
'hello'
>>>
```

讨论

这些 `strip()` 方法在读取和清理数据以备后续处理的时候是经常会被用到的。比如，你可以用它们来去掉空格，引号和完成其他任务。

但是需要注意的是去除操作不会对字符串的中间的文本产生任何影响。比如：

```
>>> s = ' hello    world \n'
>>> s = s.strip()
>>> s
'hello    world'
>>>
```

如果你想处理中间的空格，那么你需要求助其他技术。比如使用 `replace()` 方法或者是用正则表达式替换。示例如下：

```
>>> s.replace(' ', '')
'helloworld'
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>
```

通常情况下你想将字符串 `strip` 操作和其他迭代操作相结合，比如从文件中读取多行数据。如果是这样的话，那么生成器表达式就可以大显身手了。比如：

```
with open(filename) as f:
    lines = (line.strip() for line in f)
    for line in lines:
        print(line)
```

在这里，表达式 `lines = (line.strip() for line in f)` 执行数据转换操作。这种方式非常高效，因为它不需要预先读取所有数据放到一个临时的列表中去。它仅仅只是创建一个生成器，并且每次返回行之前会先执行`strip`操作。

对于更高阶的`strip`，你可能需要使用 `translate()` 方法。请参阅下一节了解更多关于字符串清理的内容。

2.12 审查清理文本字符串

问题

一些无聊的幼稚黑客在你的网站页面表单中输入文本“pýthöñ”，然后你想将这些字符清理干净。

解决方案

文本清理问题会涉及到包括文本解析与数据处理等一系列问题。在非常简单的情形下，你可能会选择使用字符串函数(比如 `str.upper()` 和 `str.lower()`)将文本转为标准格式。使用 `str.replace()` 或者 `re.sub()` 的简单替换操作能删除或者改变指定的字符序列。你同样还可以使用2.9小节的 `unicodedata.normalize()` 函数将`unicode`文本标准化。

然后，有时候你可能还想在清理操作上更进一步。比如，你可能想消除整个区间上的字符或者去除变音符。为了这样做，你可以使用经常会被忽视的 `str.translate()` 方法。为了演示，假设你现在有下面这个凌乱的字符串：

```
>>> s = 'pýthöñ\x0c\x0a\tawesome\r\n'
>>> s
'pýthöñ\x0c\x0a\tawesome\r\n'
>>>
```

第一步是清理空白字符。为了这样做，先创建一个小的转换表格然后使用 `translate()` 方法：


```

>>> remap = {
...     ord('\t') : ' ',
...     ord('\f') : ' ',
...     ord('\n') : None # Deleted
... }
>>> a = s.translate(remap)
>>> a
'pýthöñ is awesome\n'
>>>

```

正如你看的那样，空白字符 `\t` 和 `\f` 已经被重新映射到一个空格。回车字符 `r` 直接被删除。

你可以以这个表格为基础进一步构建更大的表格。比如，让我们删除所有的重音符：

```

>>> import unicodedata
>>> import sys
>>> cmb_chrs = dict.fromkeys(c for c in range(sys.maxunicode)
...                          if unicodedata.combining(chr(c)))
...
>>> b = unicodedata.normalize('NFD', a)
>>> b
'pýthöñ is awesome\n'
>>> b.translate(cmb_chrs)
'python is awesome\n'
>>>

```

上面例子中，通过使用 `dict.fromkeys()` 方法构造一个字典，每个Unicode和音符作为键，对于的值全部为 `None`。

然后使用 `unicodedata.normalize()` 将原始输入标准化为分解形式字符。然后再调用 `translate` 函数删除所有重音符。同样的技术也可以被用来删除其他类型的字符(比如控制字符等)。

作为另一个例子，这里构造一个将所有Unicode数字字符映射到对应的ASCII字符上的表格：

```

>>> digitmap = { c: ord('0') + unicodedata.digit(chr(c))
...               for c in range(sys.maxunicode)
...               if unicodedata.category(chr(c)) == 'Nd' }
...
>>> len(digitmap)
460
>>> # Arabic digits
>>> x = '\u0661\u0662\u0663'
>>> x.translate(digitmap)
'123'
>>>

```

另一种清理文本的技术设计到I/O解码与编码函数。这里的思路是先对文本做一些初步的清理，然后再结合 `encode()` 或者 `decode()` 操作来清除或修改它。比如：

```

>>> a
'pýthöñ is awesome\n'
>>> b = unicodedata.normalize('NFD', a)
>>> b.encode('ascii', 'ignore').decode('ascii')
'python is awesome\n'
>>>

```

这里的标准化操作将原来的文本分解为单独的和音符。接下来的ASCII编码/解码只是简单的一下子丢弃掉那些字符。当然，这种方法仅仅只在最后的目标就是获取到文本对应ASCII表示的时候生效。

讨论

文本字符清理一个最主要的问题应该是运行的性能。一般来讲，代码越简单运行越快。对于简单的替换操作，`str.replace()` 方法通常是最快的，甚至在你需要多次调用的时候。比如，为了清理空白字符，你可以这样做：

```

def clean_spaces(s):
    s = s.replace('\r', '')
    s = s.replace('\t', ' ')
    s = s.replace('\f', ' ')
    return s

```

如果你去测试的话，你就会发现这种方式会比使用 `translate()` 或者正则表达式要快很多。

另一方面，如果你需要执行任何复杂字符对字符的重新映射或者删除操作的话，

`translate()` 方法会非常的快。

从大的方面来讲，对于你的应用程序来说性能是你不得不去自己研究的东西。不幸的是，我们不可能给你建议一个特定的技术，使它能够适应所有的情况。因此实际情况中需要你自己去尝试不同的方法并评估它。

尽管这一节集中讨论的是文本，但是类似的技术也可以适用于字节，包括简单的替换，转换和正则表达式。

2.13 字符串对齐

问题

你想通过某种对齐方式来格式化字符串

解决方案

对于基本的字符串对齐操作，可以使用字符串的 `ljust()`，`rjust()` 和 `center()` 方法。比如：

```
>>> text = 'Hello World'
>>> text.ljust(20)
'Hello World      '
>>> text.rjust(20)
'      Hello World'
>>> text.center(20)
'    Hello World    '
>>>
```

所有这些方法都能接受一个可选的填充字符。比如：

```
>>> text.rjust(20, '=')
'====Hello World'
>>> text.center(20, '*')
'****Hello World****'
>>>
```

函数 `format()` 同样可以用来很容易的对齐字符串。你要做的就是使用 `<`，`>` 或者 `^` 字符后面紧跟一个指定的宽度。比如：

```
>>> format(text, '>20')
'      Hello World'
>>> format(text, '<20')
'Hello World      '
>>> format(text, '^20')
'  Hello World  '
>>>
```

如果你想指定一个非空格的填充字符，将它写到对齐字符的前面即可：

```
>>> format(text, '=>20s')
'====Hello World'
>>> format(text, '*^20s')
'****Hello World*****'
>>>
```

当格式化多个值的时候，这些格式代码也可以被用在 `format()` 方法中。比如：

```
>>> '{:>10s} {:>10s}'.format('Hello', 'World')
'      Hello      World'
>>>
```

`format()` 函数的一个好处是它不仅适用于字符串。它可以用来格式化任何值，使得它非常的通用。比如，你可以用它来格式化数字：

```
>>> x = 1.2345
>>> format(x, '>10')
'      1.2345'
>>> format(x, '^10.2f')
'  1.23  '
>>>
```

讨论

在老的代码中，你经常会看到被用来格式化文本的 `%` 操作符。比如：

```
>>> '%-20s' % text
'Hello World          '
>>> '%20s' % text
'          Hello World'
>>>
```

但是，在新版本代码中，你应该优先选择 `format()` 函数或者方法。`format()` 要比 `%` 操作符的功能更为强大。并且 `format()` 也比使用 `ljust()`，`rjust()` 或 `center()` 方法更通用，因为它可以用来格式化任意对象，而不仅仅是字符串。

如果想要完全了解 `format()` 函数的有用特性，请参考 [在线Python文档](#)

2.14 合并拼接字符串

问题

你想将几个小的字符串合并为一个大的字符串

解决方案

如果你想要合并的字符串是在一个序列或者 `iterable` 中，那么最快的方式就是使用 `join()` 方法。比如：

```
>>> parts = ['Is', 'Chicago', 'Not', 'Chicago?']
>>> ' '.join(parts)
'Is Chicago Not Chicago?'
>>> ','.join(parts)
'Is,Chicago,Not,Chicago?'
>>> ''.join(parts)
'IsChicagoNotChicago?'
>>>
```

初看起来，这种语法看上去会比较怪，但是 `join()` 被指定为字符串的一个方法。这样做的部分原因是你想去连接的对象可能来自各种不同的数据序列(比如列表，元组，字典，文件，集合或生成器等)，如果在所有这些对象上都定义一个 `join()` 方法明显是冗余的。因此你只需要指定你想要的分割字符串并调用他的 `join()` 方法去将文本片段组合起来。

如果你仅仅只是合并少数几个字符串，使用加号(+)通常已经足够了：

```
>>> a = 'Is Chicago'
>>> b = 'Not Chicago?'
>>> a + ' ' + b
'Is Chicago Not Chicago?'
>>>
```

加号(+)操作符在作为一些复杂字符串格式化的替代方案的时候通常也工作的很好，比如：

```
>>> print('{} {}'.format(a,b))
Is Chicago Not Chicago?
>>> print(a + ' ' + b)
Is Chicago Not Chicago?
>>>
```

如果你想在源码中将两个字面字符串合并起来，你只需要简单的将它们放到一起，不需要用加号(+). 比如：

```
>>> a = 'Hello' 'World'
>>> a
'HelloWorld'
>>>
```

讨论

字符串合并可能看上去并不需要用一整节来讨论。但是不应该小看这个问题，程序员通常在字符串格式化的时候因为选择不当而给应用程序带来严重性能损失。

最重要的需要引起注意的是，当我们使用加号(+)操作符去连接大量的字符串的时候是非常低效率的，因为加号连接会引起内存复制以及垃圾回收操作。特别的，你永远都不应像下面这样写字符串连接代码：

```
s = ''
for p in parts:
    s += p
```

这种写法会比使用 `join()` 方法运行的要慢一些，因为每一次执行+=操作的时候会创建一个新的字符串对象。你最好是先收集所有的字符串片段然后再将它们连接起来。

一个相对比较聪明的技巧是利用生成器表达式(参考1.19小节)转换数据为字符串的同时合并字符串，比如：

```
>>> data = ['ACME', 50, 91.1]
>>> ','.join(str(d) for d in data)
'ACME,50,91.1'
>>>
```

同样还得注意不必要的字符串连接操作。有时候程序员在没有必要做连接操作的时候仍然多此一举。比如在打印的时候：

```
print(a + ':' + b + ':' + c) # Ugly
print('.'.join([a, b, c])) # Still ugly
print(a, b, c, sep=':') # Better
```

当混合使用I/O操作和字符串连接操作的时候，有时候需要仔细研究你的程序。比如，考虑下面的两端代码片段：

```
# Version 1 (string concatenation)
f.write(chunk1 + chunk2)

# Version 2 (separate I/O operations)
f.write(chunk1)
f.write(chunk2)
```

如果两个字符串很小，那么第一个版本性能会更好些，因为I/O系统调用天生就慢。另外一方面，如果两个字符串很大，那么第二个版本可能会更加高效，因为它避免了创建一个很大的临时结果并且要复制大量的内存块数据。还是那句话，有时候是需要根据你的应用程序特点来决定应该使用哪种方案。

最后谈一下，如果你准备编写构建大量小字符串的输出代码，你最好考虑下使用生成器函数，利用yield语句产生输出片段。比如：

```
def sample():
    yield 'Is'
    yield 'Chicago'
    yield 'Not'
    yield 'Chicago?'
```

这种方法一个有趣的方面是它并没有对输出片段到底要怎样组织做出假设。例如，你可以简单的使用 `join()` 方法将这些片段合并起来：

```
text = ''.join(sample())
```

或者你也可以将字符串片段重定向到I/O：

```
for part in sample():
    f.write(part)
```

再或者你还可以写出一些结合I/O操作的混合方案：

```
def combine(source, maxsize):
    parts = []
    size = 0
    for part in source:
        parts.append(part)
        size += len(part)
        if size > maxsize:
            yield ''.join(parts)
            parts = []
            size = 0
    yield ''.join(parts)

# 结合文件操作
with open('filename', 'w') as f:
    for part in combine(sample(), 32768):
        f.write(part)
```

这里的关键点在于原始的生成器函数并不需要知道使用细节，它只负责生成字符串片段就行了。

2.15 字符串中插入变量

问题

你想创建一个内嵌变量的字符串，变量被它的值所表示的字符串替换掉。

解决方案

Python并没有对在字符串中简单替换变量值提供直接的支持。但是通过使用字符串的

`format()` 方法来解决这个问题。比如：

```
>>> s = '{name} has {n} messages.'
>>> s.format(name='Guido', n=37)
'Guido has 37 messages.'
>>>
```

或者，如果要被替换的变量能在变量域中找到，那么你可以结合使用 `format_map()` 和

`vars()`。就像下面这样：

```
>>> name = 'Guido'
>>> n = 37
>>> s.format_map(vars())
'Guido has 37 messages.'
>>>
```

`vars()` 还有一个有意思的特性就是它也适用于对象实例。比如：

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
>>> a = Info('Guido', 37)
>>> s.format_map(vars(a))
'Guido has 37 messages.'
>>>
```

`format` 和 `format_map()` 的一个缺陷就是它们并不能很好的处理变量缺失的情况，比如：

```
>>> s.format(name='Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

一种避免这种错误的方法是另外定义一个含有 `__missing__()` 方法的字典对象，就像下面这样：

```
class safesub(dict):
    """防止key找不到"""
    def __missing__(self, key):
        return '{' + key + '}'
```

现在你可以利用这个类包装输入后传递给 `format_map()` :

```
>>> del n # Make sure n is undefined
>>> s.format_map(safesub(vars()))
'Guido has {n} messages.'
>>>
```

如果你发现自己在代码中频繁的执行这些步骤，你可以将变量替换步骤用一个工具函数封装起来。就像下面这样：

```
import sys

def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

现在你可以像下面这样写了：

```
>>> name = 'Guido'
>>> n = 37
>>> print(sub('Hello {name}'))
Hello Guido
>>> print(sub('You have {n} messages.'))
You have 37 messages.
>>> print(sub('Your favorite color is {color}'))
Your favorite color is {color}
>>>
```

讨论

多年以来由于Python缺乏对变量替换的内置支持而导致了各种不同的解决方案。作为本节中展示的一个可能的解决方案，你可以有时候会看到像下面这样的字符串格式化代码：

```
>>> name = 'Guido'
>>> n = 37
>>> '%(name) has %(n) messages.' % vars()
'Guido has 37 messages.'
>>>
```

你可能还会看到字符串模板的使用：

```
>>> import string
>>> s = string.Template('$name has $n messages.')
>>> s.substitute(vars())
'Guido has 37 messages.'
>>>
```

然而，`format()` 和 `format_map()` 相比较上面这些方案而已更加先进，因此应该被优先选择。使用 `format()` 方法还有一个好处就是你可以获得对字符串格式化的所有支持(对齐，填充，数字格式化等待)，而这些特性是使用像模板字符串之类的方案不可能获得的。

本机还部分介绍了一些高级特性。映射或者字典类中鲜为人知的 `__missing__()` 方法可以让你定义如何处理缺失的值。在 `SafeSub` 类中，这个方法被定义为对缺失的值返回一个占位符。你可以发现缺失的值会出现在结果字符串中(在调试的时候可能很有用)，而不是产生一个 `KeyError` 异常。

`sub()` 函数使用 `sys._getframe(1)` 返回调用者的栈帧。可以从中访问属性 `f_locals` 来获得局部变量。毫无疑问绝大部分情况下在代码中去直接操作栈帧应该是不推荐的。但是，对于像字符串替换工具函数而言它是非常有用的。另外，值得注意的是 `f_locals` 是一个复制调用函数的本地变量的字典。尽管你可以改变 `f_locals` 的内容，但是这个修改对于后面的变量访问没有任何影响。所以，虽说访问一个栈帧看上去很邪恶，但是对它的任何操作不会覆盖和改变调用者本地变量的值。

2.16 以指定列宽格式化字符串

问题

你有一些长字符串，想以指定的列宽将它们重新格式化。

解决方案

使用 `textwrap` 模块来格式化字符串的输出。比如，假如你有下列的长字符串：

```
s = "Look into my eyes, look into my eyes, the eyes, the eyes, \
the eyes, not around the eyes, don't look around the eyes, \
look into my eyes, you're under."
```

下面演示使用 `textwrap` 格式化字符串的多种方式：

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes, don't look around the eyes, look into my eyes,
you're under.

>>> print(textwrap.fill(s, 40))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not around
the eyes, don't look around the eyes,
look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, initial_indent='    '))
    Look into my eyes, look into my
eyes, the eyes, the eyes, the eyes, not
around the eyes, don't look around the
eyes, look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, subsequent_indent='    '))
Look into my eyes, look into my eyes,
    the eyes, the eyes, the eyes, not
    around the eyes, don't look around
    the eyes, look into my eyes, you're
    under.
```

讨论

`textwrap` 模块对于字符串打印是非常有用的，特别是当你希望输出自动匹配终端大小的时候。你可以使用 `os.get_terminal_size()` 方法来获取终端的大小尺寸。比如：

```
>>> import os
>>> os.get_terminal_size().columns
80
>>>
```

`fill()` 方法接受一些其他可选参数来控制tab, 语句结尾等。参阅 [textwrap.TextWrapper文档](#) 获取更多内容。

2.17 在字符串中处理html和xml

问题

你想将HTML或者XML实体如 `&entity;` 或 `&#code;` 替换为对应的文本。再者, 你需要转换文本中特定的字符(比如<, >, 或 &)。

解决方案

如果你想替换文本字符串中的‘<’或者‘>’, 使用 `html.escape()` 函数可以很容易的完成。比如:

```
>>> s = 'Elements are written as "<tag>text</tag>".'  
>>> import html  
>>> print(s)  
Elements are written as "<tag>text</tag>".  
>>> print(html.escape(s))  
Elements are written as "&quot;&lt;tag&gt;text&lt;/tag&gt;&quot;".  
  
>>> # Disable escaping of quotes  
>>> print(html.escape(s, quote=False))  
Elements are written as "&lt;tag&gt;text&lt;/tag&gt;".  
>>>
```

如果你正在处理的是ASCII文本, 并且想将非ASCII文本对应的编码实体嵌入进去, 可以给某些I/O函数传递参数 `errors='xmlcharrefreplace'` 来达到这个目。比如:

```
>>> s = 'Spicy Jalapeño'  
>>> s.encode('ascii', errors='xmlcharrefreplace')  
b'Spicy Jalape&#241;o'  
>>>
```

为了替换文本中的编码实体, 你需要使用另外一种方法。如果你正在处理HTML或者XML文本, 试着先使用一个合适的HTML或者XML解析器。通常情况下, 这些工具会自动替换这些编码值, 你无需担心。

有时候, 如果你接收到了一些含有编码值的原始文本, 需要手动去做替换, 通常你只需要使用HTML或者XML解析器的一些相关工具函数/方法即可。比如:

```
>>> s = 'Spicy &quot;Jalape&#241;o&quot;.'
>>> from html.parser import HTMLParser
>>> p = HTMLParser()
>>> p.unescape(s)
'Spicy "Jalapeño".'
>>>
>>> t = 'The prompt is &gt;&gt;&gt;.'
>>> from xml.sax.saxutils import unescape
>>> unescape(t)
'The prompt is >>>'
>>>
```

讨论

在生成HTML或者XML文本的时候，如果正确的转换特殊标记字符是一个很容易被忽视的细节。特别是当你使用 `print()` 函数或者其他字符串格式化来产生输出的时候。使用像 `html.escape()` 的工具函数可以很容易的解决这类问题。

如果你想以其他方式处理文本，还有一些其他的工具函数比如

`xml.sax.saxutils.unescape()` 可以帮助你。然而，你应该先调研清楚怎样使用一个合适的解析器。比如，如果你在处理HTML或XML文本，使用某个解析模块比如 `html.parse` 或 `xml.etree.ElementTree` 已经帮你自动处理了相关的替换细节。

2.18 字符串令牌解析

问题

你有一个字符串，想从左至右将其解析为一个令牌流。

解决方案

假如你有下面这样一个文本字符串：

```
text = 'foo = 23 + 42 * 10'
```

为了令牌化字符串，你不仅需要匹配模式，还得指定模式的类型。比如，你可能想将字符串像下面这样转换为序列对：

```
tokens = [('NAME', 'foo'), ('EQ', '='), ('NUM', '23'), ('PLUS', '+'),
          ('NUM', '42'), ('TIMES', '*'), ('NUM', '10')]
```

为了执行这样的切分，第一步就是像下面这样利用命名捕获组的正则表达式来定义所有可能的令牌，包括空格：

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ = r'(?P<EQ>=)'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))
```

在上面的模式中，`?P<TOKENNAME>` 用于给一个模式命名，供后面使用。

下一步，为了令牌化，使用模式对象很少被人知道的 `scanner()` 方法。这个方法会创建一个 `scanner` 对象，在这个对象上不断的调用 `match()` 方法会一步步的扫描目标文本，每步一个匹配。下面是演示一个 `scanner` 对象如何工作的交互式例子：

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

实际使用这种技术的时候，可以很容易的像下面这样将上述代码打包到一个生成器中：

```
def generate_tokens(pat, text):
    Token = namedtuple('Token', ['type', 'value'])
    scanner = pat.scanner(text)
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Example use
for tok in generate_tokens(master_pat, 'foo = 42'):
    print(tok)

# Produces output
# Token(type='NAME', value='foo')
# Token(type='WS', value=' ')
# Token(type='EQ', value='=')
# Token(type='WS', value=' ')
# Token(type='NUM', value='42')
```

如果你想过滤令牌流，你可以定义更多的生成器函数或者使用一个生成器表达式。比如，下面演示怎样过滤所有的空白令牌：

```
tokens = (tok for tok in generate_tokens(master_pat, text)
          if tok.type != 'WS')
for tok in tokens:
    print(tok)
```

讨论

通常来讲令牌化是很多高级文本解析与处理的第一步。为了使用上面的扫描方法，你需要记住这里一些重要的几点。第一点就是你必须确认你使用正则表达式指定了所有输入中可能出现的文本序列。如果有任何不可匹配的文本出现了，扫描就会直接停止。这也是为什么上面例子中必须指定空白字符令牌的原因。

令牌的顺序也是有影响的。`re` 模块会按照指定好的顺序去做匹配。因此，如果一个模式恰好是另一个更长模式的子字符串，那么你需要确定长模式写在前面。比如：

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ])) # Correct
# master_pat = re.compile('|'.join([LT, LE, EQ])) # Incorrect
```


第二个模式是错的，因为它会将文本`<=`匹配为令牌`LT`紧跟着`EQ`，而不是单独的令牌`LE`，这个并不是我们想要的结果。

最后，你需要留意下子字符串形式的模式。比如，假设你有如下两个模式：

```
PRINT = r'(P<PRINT>print)'  
NAME = r'(P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'  
  
master_pat = re.compile('|'.join([PRINT, NAME]))  
  
for tok in generate_tokens(master_pat, 'printer'):  
    print(tok)  
  
# Outputs :  
# Token(type='PRINT', value='print')  
# Token(type='NAME', value='er')
```

关于更高阶的令牌化技术，你可能需要查看 [PyParsing](#) 或者 [PLY](#) 包。一个调用PLY的例子在下一节会有演示。

2.19 实现一个简单的递归下降分析器

问题

你想根据一组语法规则解析文本并执行命令，或者构造一个代表输入的抽象语法树。如果语法非常简单，你可以自己写这个解析器，而不是使用一些框架。

解决方案

在这个问题中，我们集中讨论根据特殊语法去解析文本的问题。为了这样做，你首先要以BNF或者EBNF形式指定一个标准语法。比如，一个简单数学表达式语法可能像下面这样：

```
expr ::= expr + term  
      | expr - term  
      | term  
  
term ::= term * factor  
      | term / factor  
      | factor  
  
factor ::= ( expr )  
        | NUM
```

或者，以EBNF形式：

```
expr ::= term { (+|-) term }*
term ::= factor { (*|/) factor }*
factor ::= ( expr )
         |  NUM
```

在EBNF中，被包含在 `{...}*` 中的规则是可选的。`*`代表0次或多次重复(跟正则表达式中意义是一样的)。

现在，如果你对BNF的工作机制还不是很明白的话，就把它当做是一组左右符号可相互替换的规则。一般来讲，解析的原理就是你利用BNF完成多个替换和扩展以匹配输入文本和语法规则。为了演示，假设你正在解析形如 `3 + 4 * 5` 的表达式。这个表达式先要通过使用2.18节中介绍的技术分解为一组令牌流。结果可能是像下列这样的令牌序列：

```
NUM + NUM * NUM
```

在此基础上，解析动作会试着去通过替换操作匹配语法到输入令牌：

```
expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*
expr ::= NUM + NUM * NUM
```

下面所有的解析步骤可能需要花点时间弄明白，但是它们原理都是查找输入并试着去匹配语法规则。第一个输入令牌是NUM，因此替换首先会匹配那个部分。一旦匹配成功，就会进入下一个令牌+，以此类推。当已经确定不能匹配下一个令牌的时候，右边的部分(比如 `{ (*|/) factor }*`)就会被清理掉。在一个成功的解析中，整个右边部分会完全展开来匹配输入令牌流。

有了前面的知识背景，下面我们举一个简单示例来展示如何构建一个递归下降表达式求值程序：

```

#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 下降解析器
Desc :
"""

import re
import collections

# Token specification
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
MINUS = r'(?P<MINUS>-)'
TIMES = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\()'
RPAREN = r'(?P<RPAREN>\))'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))

# Tokenizer
Token = collections.namedtuple('Token', ['type', 'value'])

def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok

# Parser
class ExpressionEvaluator:
    """
    Implementation of a recursive descent parser. Each method
    implements a single grammar rule. Use the ._accept() method
    to test and accept the current lookahead token. Use the ._expect()
    method to exactly match and discard the next token on on the input
    (or raise a SyntaxError if it doesn't match).
    """

    def parse(self, text):
        self.tokens = generate_tokens(text)
        self.tok = None # Last symbol consumed
        self.nexttok = None # Next symbol tokenized
        self._advance() # Load first lookahead token
        return self.expr()

    def _advance(self):
        'Advance one token ahead'
        self.tok, self.nexttok = self.nexttok, next(self.tokens, None)

    def _accept(self, toktype):
        'Test and consume the next token if it matches toktype'
        if self.nexttok and self.nexttok.type == toktype:

```

```

        self._advance()
        return True
    else:
        return False

def _expect(self, toktype):
    'Consume next token if it matches toktype or raise SyntaxError'
    if not self._accept(toktype):
        raise SyntaxError('Expected ' + toktype)

# Grammar rules follow
def expr(self):
    "expression ::= term { ('+'|'-') term }*"
    exprval = self.term()
    while self._accept('PLUS') or self._accept('MINUS'):
        op = self.tok.type
        right = self.term()
        if op == 'PLUS':
            exprval += right
        elif op == 'MINUS':
            exprval -= right
    return exprval

def term(self):
    "term ::= factor { ('*'|'/') factor }*"
    termval = self.factor()
    while self._accept('TIMES') or self._accept('DIVIDE'):
        op = self.tok.type
        right = self.factor()
        if op == 'TIMES':
            termval *= right
        elif op == 'DIVIDE':
            termval /= right
    return termval

def factor(self):
    "factor ::= NUM | ( expr )"
    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
    else:
        raise SyntaxError('Expected NUMBER or LPAREN')

def descent_parser():
    e = ExpressionEvaluator()
    print(e.parse('2'))
    print(e.parse('2 + 3'))
    print(e.parse('2 + 3 * 4'))
    print(e.parse('2 + (3 + 4) * 5'))
    # print(e.parse('2 + (3 + * 4)'))
    # Traceback (most recent call last):
    #   File "<stdin>", line 1, in <module>
    #   File "exprparse.py", line 40, in parse
    #   return self.expr()

```

```
# File "exprparse.py", line 67, in expr
# right = self.term()
# File "exprparse.py", line 77, in term
# termval = self.factor()
# File "exprparse.py", line 93, in factor
# exprval = self.expr()
# File "exprparse.py", line 67, in expr
# right = self.term()
# File "exprparse.py", line 77, in term
# termval = self.factor()
# File "exprparse.py", line 97, in factor
# raise SyntaxError("Expected NUMBER or LPAREN")
# SyntaxError: Expected NUMBER or LPAREN
```

```
if __name__ == '__main__':
    descent_parser()
```

讨论

文本解析是一个很大的主题，一般会占用学生学习编译课程时刚开始的三周时间。如果你在找寻关于语法，解析算法等相关的背景知识的话，你应该去看一下编译器书籍。很显然，关于这方面的内容太多，不可能在这里全部展开。

尽管如此，编写一个递归下降解析器的整体思路是比较简单的。开始的时候，你先获得所有的语法规则，然后将其转换为一个函数或者方法。因此如果你的语法类似这样：

```
expr ::= term { ('+'|'-') term }*
term ::= factor { ('*'|'/') factor }*
factor ::= '(' expr ')'
         | NUM
```

你应该首先将它们转换成一组像下面这样的方法：

```
class ExpressionEvaluator:
    ...
    def expr(self):
    ...
    def term(self):
    ...
    def factor(self):
    ...
```

每个方法要完成的任务很简单 - 它必须从左至右遍历语法规则的每一部分，处理每个令牌。从某种意义上讲，方法的目的是要么处理完语法规则，要么产生一个语法错误。为了这样做，需采用下面的这些实现方法：

- 如果规则中的下个符号是另外一个语法规则的名字(比如term或factor)，就简单的调用同名的方法即可。这就是该算法中“下降”的由来 - 控制下降到另一个语法规则中去。有时候规则会调用已经执行的方法(比如，在 `factor ::= '('expr ')'` 中对expr的调用)。这就是算法中“递归”的由来。
- 如果规则中下一个符号是个特殊符号(比如()，你得查找下一个令牌并确认是一个精确匹配)。如果不匹配，就产生一个语法错误。这一节中的 `_expect()` 方法就是用来做这一步的。
- 如果规则中下一个符号为一些可能的选择项(比如+或-)，你必须对每一种可能情况检查下一个令牌，只有当它匹配一个的时候才能继续。这也是本节示例中 `_accept()` 方法的目的。它相当于`_expect()`方法的弱化版本，因为如果一个匹配找到了它会继续，但是如果没找到，它不会产生错误而是回滚(允许后续的检查继续进行)。
- 对于有重复部分的规则(比如在规则表达式 `::= term { ('+'|'-') term }*` 中)，重复动作通过一个while循环来实现。循环主体会收集或处理所有的重复元素直到没有其他元素可以找到。
- 一旦整个语法规则处理完成，每个方法会返回某种结果给调用者。这就是在解析过程中值是怎样累加的原理。比如，在表达式求值程序中，返回值代表表达式解析后的部分结果。最后所有值会在最顶层的语法规则方法中合并起来。

尽管向你演示的是一个简单的例子，递归下降解析器可以用来实现非常复杂的解析。比如，Python语言本身就是通过一个递归下降解析器去解释的。如果你对此感兴趣，你可以通过查看Python源码文件Grammar/Grammar来研究下底层语法机制。看完你会发现，通过手动方式去实现一个解析器其实会有很多的局限和不足之处。

其中一个局限就是它们不能被用于包含任何左递归的语法规则中。比如，加入你需要翻译下面这样一个规则：

```
items ::= items ',' item
        | item
```

为了这样做，你可能会像下面这样使用 `items()` 方法：

```
def items(self):
    itemsval = self.items()
    if itemsval and self._accept(','):
        itemsval.append(self.item())
    else:
        itemsval = [ self.item() ]
```

唯一的问题是这个方法根本不能工作，事实上，它会产生一个无限递归错误。

关于语法规则本身你可能也会碰到一些棘手的问题。比如，你可能想知道下面这个简单扼语法是否表述得当：

```
expr ::= factor { ('+'|'-'|'*'|'/') factor }*

factor ::= '(' expression ')'
         | NUM
```

这个语法看上去没啥问题，但是它却不能察觉到标准四则运算中的运算符优先级。比如，表达式 `"3 + 4 * 5"` 会得到35而不是期望的23。分开使用“expr”和“term”规则可以让它正确的工作。

对于复杂的语法，你最好是选择某个解析工具比如PyParsing或者是PLY。下面是使用PLY来重写表达式求值程序的代码：

```
from ply.lex import lex
from ply.yacc import yacc

# Token List
tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN' ]
# Ignored characters
t_ignore = ' \t\n'
# Token specifications (as regexs)
t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Token processing functions
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Error handler
def t_error(t):
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Build the Lexer
lexer = lex()

# Grammar rules and handler functions
def p_expr(p):
    '''
    expr : expr PLUS term
```

```

        | expr MINUS term
    ...
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]

def p_expr_term(p):
    ...

    expr : term
    ...

    p[0] = p[1]

def p_term(p):
    ...

    term : term TIMES factor
        | term DIVIDE factor
    ...

    if p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]

def p_term_factor(p):
    ...

    term : factor
    ...

    p[0] = p[1]

def p_factor(p):
    ...

    factor : NUM
    ...

    p[0] = p[1]

def p_factor_group(p):
    ...

    factor : LPAREN expr RPAREN
    ...

    p[0] = p[2]

def p_error(p):
    print('Syntax error')

parser = yacc()

```

这个程序中，所有代码都位于一个比较高的层次。你只需要为令牌写正则表达式和规则匹配时的高阶处理函数即可。而实际的运行解析器，接受令牌等等底层动作已经被库函数实现了。

下面是一个怎样使用得到的解析对象的例子：


```
>>> parser.parse('2')
2
>>> parser.parse('2+3')
5
>>> parser.parse('2+(3+4)*5')
37
>>>
```

如果你想在你的编程过程中来点挑战和刺激，编写解析器和编译器是个不错的选择。再次，一本编译器的书籍会包含很多底层的理论知识。不过很多好的资源也可以在网上找到。Python自己的ast模块也值得去看一下。

2.20 字节字符串上的字符串操作

问题

你想在字节字符串上执行普通的文本操作(比如移除，搜索和替换)。

解决方案

字节字符串同样也支持大部分和文本字符串一样的内置操作。比如：

```
>>> data = b'Hello World'
>>> data[0:5]
b'Hello'
>>> data.startswith(b'Hello')
True
>>> data.split()
[b'Hello', b'World']
>>> data.replace(b'Hello', b'Hello Cruel')
b'Hello Cruel World'
>>>
```

这些操作同样也适用于字节数组。比如：

```
>>> data = bytearray(b'Hello World')
>>> data[0:5]
bytearray(b'Hello')
>>> data.startswith(b'Hello')
True
>>> data.split()
[bytearray(b'Hello'), bytearray(b'World')]
>>> data.replace(b'Hello', b'Hello Cruel')
bytearray(b'Hello Cruel World')
>>>
```

你可以使用正则表达式匹配字节字符串，但是正则表达式本身必须也是字节串。比如：

```
>>>
>>> data = b'FOO:BAR,SPAM'
>>> import re
>>> re.split('[:,]',data)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/local/lib/python3.3/re.py", line 191, in split
return _compile(pattern, flags).split(string, maxsplit)
TypeError: can't use a string pattern on a bytes-like object
>>> re.split(b'[:,]',data) # Notice: pattern as bytes
[b'FOO', b'BAR', b'SPAM']
>>>
```

讨论

大多数情况下，在文本字符串上的操作均可用于字节字符串。然而，这里也有一些需要注意的不同点。首先，字节字符串的索引操作返回整数而不是单独字符。比如：

```
>>> a = 'Hello World' # Text string
>>> a[0]
'H'
>>> a[1]
'e'
>>> b = b'Hello World' # Byte string
>>> b[0]
72
>>> b[1]
101
>>>
```

这种语义上的区别会对于处理面向字节的字符数据有影响。

第二点，字节字符串不会提供一个美观的字符串表示，也不能很好的打印出来，除非它们先被解码为一个文本字符串。比如：

```
>>> s = b'Hello World'
>>> print(s)
b'Hello World' # Observe b'...'
>>> print(s.decode('ascii'))
Hello World
>>>
```

类似的，也不存在任何适用于字节字符串的格式化操作：

```
>>> b'%10s %10d %10.2f' % (b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'
>>> b'{} {} {}'.format(b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'format'
>>>
```

如果你想格式化字节字符串，你得先使用标准的文本字符串，然后将其编码为字节字符串。比如：

```
>>> '{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('ascii')
b'ACME 100 490.10'
>>>
```

最后需要注意的是，使用字节字符串可能会改变一些操作的语义，特别是那些跟文件系统有关的操作。比如，如果你使用一个编码为字节的文件名，而不是一个普通的文本字符串，会禁用文件名的编码/解码。比如：

```
>>> # Write a UTF-8 filename
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('spicy')
...
>>> # Get a directory listing
>>> import os
>>> os.listdir('.') # Text string (names are decoded)
['jalape\u00f1o.txt']
>>> os.listdir(b'.') # Byte string (names left as bytes)
[b'jalapen\xcc\x83o.txt']
>>>
```

注意例子中的最后部分给目录名传递一个字节字符串是怎样导致结果中文件名以未解码字节返回的。在目录中的文件名包含原始的UTF-8编码。参考5.15小节获取更多文件名相关的内容。

最后提一点，一些程序员为了提升程序执行的速度会倾向于使用字节字符串而不是文本字符串。尽管操作字节字符串确实会比文本更加高效(因为处理文本固有的Unicode相关开销)。这样做通常会导致非常杂乱的代码。你会经常发现字节字符串并不能和Python的其他部分工作的很好，并且你还得手动处理所有的编码/解码操作。坦白讲，如果你在处理文本的话，就直接在程序中使用普通的文本字符串而不是字节字符串。不做死就不会死！

第三章：数字日期和时间

在Python中执行整数和浮点数的数学运算时很简单的。尽管如此，如果你需要执行分数、数组或者是日期和时间的运算的话，就得做更多的工作了。本章集中讨论的就是这些主题。

Contents:

3.1 数字的四舍五入

问题

你想对浮点数执行指定精度的舍入运算。

解决方案

对于简单的舍入运算，使用内置的 `round(value, ndigits)` 函数即可。比如：

```
>>> round(1.23, 1)
1.2
>>> round(1.27, 1)
1.3
>>> round(-1.27, 1)
-1.3
>>> round(1.25361,3)
1.254
>>>
```

当一个值刚好在两个边界的中间的时候，`round` 函数返回离它最近的偶数。也就是说，对1.5或者2.5的舍入运算都会得到2。

传给 `round()` 函数的 `ndigits` 参数可以是负数，这种情况下，舍入运算会作用在十位、百位、千位等上面。比如：

```
>>> a = 1627731
>>> round(a, -1)
1627730
>>> round(a, -2)
1627700
>>> round(a, -3)
1628000
>>>
```

讨论

不要将舍入和格式化输出搞混淆了。如果你的目的只是简单的输出一定宽度的数，你不需要使用 `round()` 函数。而仅仅只需要在格式化的时候指定精度即可。比如：

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'value is {:.3f}'.format(x)
'value is 1.235'
>>>
```

同样，不要试着去舍入浮点值来“修正”表面上看起来正确的问题。比如，你可能倾向于这样做：

```
>>> a = 2.1
>>> b = 4.2
>>> c = a + b
>>> c
6.3000000000000001
>>> c = round(c, 2) # "Fix" result (???)
>>> c
6.3
>>>
```

对于大多数使用到浮点的程序，没有必要也不推荐这样做。尽管在计算的时候会有一点点小的误差，但是这些小的误差是能被理解与容忍的。如果不能允许这样的小误差(比如涉及到金融领域)，那么就得考虑使用 `decimal` 模块了，下一节我们会详细讨论。

3.2 执行精确的浮点数运算

问题

你需要对浮点数执行精确的计算操作，并且不希望有任何小误差的出现。

解决方案

浮点数的一个普遍问题是它们并不能精确的表示十进制数。并且，即使是最简单的数学运算也会产生小的误差，比如：

```
>>> a = 4.2
>>> b = 2.1
>>> a + b
6.3000000000000001
>>> (a + b) == 6.3
False
>>>
```

这些错误是由底层CPU和IEEE 754标准通过自己的浮点单位去执行算术时的特征。由于Python的浮点数据类型使用底层表示存储数据，因此你没办法去避免这样的误差。

如果你想更加精确(并能容忍一定的性能损耗)，你可以使用 `decimal` 模块：

```
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
```

初看起来，上面的代码好像有点奇怪，比如我们用字符串来表示数字。然而，`Decimal` 对象会像普通浮点数一样的工作(支持所有的常用数学运算)。如果你打印它们或者在字符串格式化函数中使用它们，看起来跟普通数字没什么两样。

`decimal` 模块的一个主要特征是允许你控制计算的每一方面，包括数字位数和四舍五入运算。为了这样做，你先得创建一个本地上下文并更改它的设置，比如：

```
>>> from decimal import localcontext
>>> a = Decimal('1.3')
>>> b = Decimal('1.7')
>>> print(a / b)
0.7647058823529411764705882353
>>> with localcontext() as ctx:
...     ctx.prec = 3
...     print(a / b)
...
0.765
>>> with localcontext() as ctx:
...     ctx.prec = 50
...     print(a / b)
...
0.76470588235294117647058823529411764705882352941176
>>>
```

讨论

`decimal` 模块实现了IBM的“通用小数运算规范”。不用说，有很多的配置选项这本书没有提到。

Python新手会倾向于使用 `decimal` 模块来处理浮点数的精确运算。然而，先理解你的应用程序目的是非常重要的。如果你是在做科学计算或工程领域的计算、电脑绘图，或者是科学领域的大多数运算，那么使用普通的浮点类型是比较普遍的做法。其中一个原因是，在真实世界中很少会要求精确到普通浮点数能提供的17位精度。因此，计算过程中的那么一点点的误差是被允许的。第二点就是，原生的浮点数计算要快的多-有时候你在执行大量运算的时候速度也是非常重要的。

即便如此，你却不能完全忽略误差。数学家花了大量时间去研究各类算法，有些处理误差会比其他方法更好。你也得注意下减法删除已经大数和小数的加分运算所带来的影响。比如：

```
>>> nums = [1.23e+18, 1, -1.23e+18]
>>> sum(nums) # Notice how 1 disappears
0.0
>>>
```

上面的错误可以利用 `math.fsum()` 所提供的更精确计算能力来解决：

```
>>> import math
>>> math.fsum(nums)
1.0
>>>
```

然而，对于其他的算法，你应该仔细研究它并理解它的误差产生来源。

总的来说，`decimal` 模块主要用在涉及到金融的领域。在这类程序中，哪怕是一点小小的误差在计算过程中蔓延都是不允许的。因此，`decimal` 模块为解决这类问题提供了方法。当Python和数据库打交道的时候也通常会遇到 `Decimal` 对象，并且，通常也是在处理金融数据的时候。

3.3 数字的格式化输出

问题

你需要将数字格式化后输出，并控制数字的位数、对齐、千位分隔符和其他的细节。

解决方案

格式化输出单个数字的时候，可以使用内置的 `format()` 函数，比如：


```

>>> x = 1234.56789

>>> # Two decimal places of accuracy
>>> format(x, '0.2f')
'1234.57'

>>> # Right justified in 10 chars, one-digit accuracy
>>> format(x, '>10.1f')
'   1234.6'

>>> # Left justified
>>> format(x, '<10.1f')
'1234.6   '

>>> # Centered
>>> format(x, '^10.1f')
' 1234.6  '

>>> # Inclusion of thousands separator
>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
>>>

```

如果你想使用指数记法，将f改成e或者E(取决于指数输出的大小写形式)。比如：

```

>>> format(x, 'e')
'1.234568e+03'
>>> format(x, '0.2E')
'1.23E+03'
>>>

```

同时指定宽度和精度的一般形式是 `'[<>^]?width[,]?(.digits)?'`，其中 `width` 和 `digits` 为整数，? 代表可选部分。同样的格式也被用在字符串的 `format()` 方法中。比如：

```

>>> 'The value is {:0,.2f}'.format(x)
'The value is 1,234.57'
>>>

```

讨论

数字格式化输出通常是比较简单的。上面演示的技术同时适用于浮点数和 `decimal` 模块中的 `Decimal` 数字对象。

当指定数字的位数后，结果值会根据 `round()` 函数同样的规则进行四舍五入后返回。比如：

```
>>> x
1234.56789
>>> format(x, '0.1f')
'1234.6'
>>> format(-x, '0.1f')
'-1234.6'
>>>
```

包含千位符的格式化跟本地化没有关系。如果你需要根据地区来显示千位符，你需要自己去调查下 `locale` 模块中的函数了。你同样也可以使用字符串的 `translate()` 方法来交换千位符。比如：

```
>>> swap_separators = { ord('.'):',' , ord(','):'.' }
>>> format(x, ',').translate(swap_separators)
'1.234,56789'
>>>
```

在很多Python代码中会看到使用%来格式化数字的，比如：

```
>>> '%0.2f' % x
'1234.57'
>>> '%10.1f' % x
'      1234.6'
>>> '%-10.1f' % x
'1234.6      '
>>>
```

这种格式化方法也是可行的，不过比更加先进的 `format()` 要差一点。比如，在使用%操作符格式化数字的时候，一些特性(添加千位符)并不能被支持。

3.4 二八十六进制整数

问题

你需要转换或者输出使用二进制，八进制或十六进制表示的整数。

解决方案

为了将整数转换为二进制、八进制或十六进制的文本串，可以分别使用 `bin()`，`oct()` 或 `hex()` 函数：

```
>>> x = 1234
>>> bin(x)
'0b10011010010'
>>> oct(x)
'0o2322'
>>> hex(x)
'0x4d2'
>>>
```

另外，如果你不想输出 `0b`，`0o` 或者 `0x` 的前缀的话，可以使用 `format()` 函数。比如：

```
>>> format(x, 'b')
'10011010010'
>>> format(x, 'o')
'2322'
>>> format(x, 'x')
'4d2'
>>>
```

整数是有符号的，所以如果你在处理负数的话，输出结果会包含一个负号。比如：

```
>>> x = -1234
>>> format(x, 'b')
'-10011010010'
>>> format(x, 'x')
'-4d2'
>>>
```

如果你想产生一个无符号值，你需要增加一个指示最大位长度的值。比如为了显示32位的值，可以像下面这样写：

```
>>> x = -1234
>>> format(2**32 + x, 'b')
'1111111111111111111111111111111101100101110'
>>> format(2**32 + x, 'x')
'fffffb2e'
>>>
```

为了以不同的进制转换整数字符串，简单的使用带有进制的 `int()` 函数即可：

```
>>> int('4d2', 16)
1234
>>> int('10011010010', 2)
1234
>>>
```

讨论

大多数情况下处理二进制、八进制和十六进制整数是很简单的。只要记住这些转换属于整数和其对应的文本表示之间的转换即可。永远只有一种整数类型。

最后，使用八进制的程序员有一点需要注意下。Python指定八进制数的语法跟其他语言稍有不同。比如，如果你像下面这样指定八进制，会出现语法错误：

```
>>> import os
>>> os.chmod('script.py', 0755)
File "<stdin>", line 1
    os.chmod('script.py', 0755)
                              ^
SyntaxError: invalid token
>>>
```

需确保八进制数的前缀是 `0o`，就像下面这样：

```
>>> os.chmod('script.py', 0o755)
>>>
```

3.5 字节到大整数的打包与解包

问题

你有一个字节字符串并想将它解压成一个整数。或者，你需要将一个大整数转换为一个字节字符串。

解决方案

假设你的程序需要处理一个拥有128位长的16个元素的字节字符串。比如：

```
data = b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

为了将bytes解析为整数，使用 `int.from_bytes()` 方法，并像下面这样指定字节顺序：

```
>>> len(data)
16
>>> int.from_bytes(data, 'little')
69120565665751139577663547927094891008
>>> int.from_bytes(data, 'big')
94522842520747284487117727783387188
>>>
```

为了将一个大整数转换为一个字节字符串，使用 `int.to_bytes()` 方法，并像下面这样指定字节数和字节顺序：

```
>>> x = 94522842520747284487117727783387188
>>> x.to_bytes(16, 'big')
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> x.to_bytes(16, 'little')
b'4\x00#\x00\x01\xef\xcd\x00\xab\x90x\x00V4\x12\x00'
>>>
```

讨论

大整数和字节字符串之间的转换操作并不常见。然而，在一些应用领域有时候也会出现，比如密码学或者网络。例如，IPv6网络地址使用一个128位的整数表示。如果你要从一个数据记录中提取这样的值的时候，你就会面对这样的问题。

作为一种替代方案，你可能想使用6.11小节中所介绍的 `struct` 模块来解压字节。这样也行得通，不过利用 `struct` 模块来解压对于整数的大小是有限制的。因此，你可能想解压多个字节串并将结果合并为最终的结果，就像下面这样：

```
>>> data
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> import struct
>>> hi, lo = struct.unpack('>QQ', data)
>>> (hi << 64) + lo
94522842520747284487117727783387188
>>>
```

字节顺序规则(`little`或`big`)仅仅指定了构建整数时的字节的低位高位排列方式。我们从下面精心构造的16进制数的表示中可以很容易的看出来：

```
>>> x = 0x01020304
>>> x.to_bytes(4, 'big')
b'\x01\x02\x03\x04'
>>> x.to_bytes(4, 'little')
b'\x04\x03\x02\x01'
>>>
```

如果你试着将一个整数打包为字节字符串，那么它就不合适了，你会得到一个错误。如果需要的话，你可以使用 `int.bit_length()` 方法来决定需要多少字节位来存储这个值。

```
>>> x = 523 ** 23
>>> x
335381300113661875107536852714019056160355655333978849017944067
>>> x.to_bytes(16, 'little')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
>>> x.bit_length()
208
>>> nbytes, rem = divmod(x.bit_length(), 8)
>>> if rem:
...     nbytes += 1
...
>>>
>>> x.to_bytes(nbytes, 'little')
b'\x03X\xf1\x82iT\x96\xac\xc7c\x16\xf3\xb9\xcf...\xd0'
>>>
```

3.6 复数的数学运算

问题

你写的最新的网络认证方案代码遇到了一个难题，并且你唯一的解决办法就是使用复数空间。再或者是你仅仅需要使用复数来执行一些计算操作。

解决方案

复数可以用使用函数 `complex(real, imag)` 或者是带有后缀j的浮点数来指定。比如：

```
>>> a = complex(2, 4)
>>> b = 3 - 5j
>>> a
(2+4j)
>>> b
(3-5j)
>>>
```

对应的实部、虚部和共轭复数可以很容易的获取。就像下面这样：

```
>>> a.real
2.0
>>> a.imag
4.0
>>> a.conjugate()
(2-4j)
>>>
```

另外，所有常见的数学运算都可以工作：

```
>>> a + b
(5-1j)
>>> a * b
(26+2j)
>>> a / b
(-0.4117647058823529+0.6470588235294118j)
>>> abs(a)
4.47213595499958
>>>
```

如果要执行其他的复数函数比如正弦、余弦或平方根，使用 `cmath` 模块：

```
>>> import cmath
>>> cmath.sin(a)
(24.83130584894638-11.356612711218174j)
>>> cmath.cos(a)
(-11.36423470640106-24.814651485634187j)
>>> cmath.exp(a)
(-4.829809383269385-5.5920560936409816j)
>>>
```

讨论

Python中大部分与数学相关的模块都能处理复数。比如如果你使用 `numpy`，可以很容易的构造一个复数数组并在这个数组上执行各种操作：

```
>>> import numpy as np
>>> a = np.array([2+3j, 4+5j, 6-7j, 8+9j])
>>> a
array([ 2.+3.j,  4.+5.j,  6.-7.j,  8.+9.j])
>>> a + 2
array([ 4.+3.j,  6.+5.j,  8.-7.j, 10.+9.j])
>>> np.sin(a)
array([ 9.15449915 -4.16890696j, -56.16227422 -48.50245524j,
       -153.20827755-526.47684926j, 4008.42651446-589.49948373j])
>>>
```

Python的标准数学函数确实情况下并不能产生复数值，因此你的代码中不可能会出现复数返回值。比如：

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

如果你想生成一个复数返回结果，你必须显示的使用 `cmath` 模块，或者在某个支持复数的库中声明复数类型的使用。比如：


```
>>> import cmath
>>> cmath.sqrt(-1)
1j
>>>
```

3.7 无穷大与NaN

问题

你想创建或测试正无穷、负无穷或NaN(非数字)的浮点数。

解决方案

Python并没有特殊的语法来表示这些特殊的浮点值，但是可以使用 `float()` 来创建它们。比如：

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

为了测试这些值的存在，使用 `math.isinf()` 和 `math.isnan()` 函数。比如：

```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
>>>
```

讨论

想了解更多这些特殊浮点值的信息，可以参考IEEE 754规范。然而，也有一些地方需要你特别注意，特别是跟比较和操作符相关的时候。

无穷大数在执行数学计算的时候会传播，比如：

```
>>> a = float('inf')
>>> a + 45
inf
>>> a * 10
inf
>>> 10 / a
0.0
>>>
```

但是有些操作时未定义的并会返回一个NaN结果。比如：

```
>>> a = float('inf')
>>> a/a
nan
>>> b = float('-inf')
>>> a + b
nan
>>>
```

NaN值会在所有操作中传播，而不会产生异常。比如：

```
>>> c = float('nan')
>>> c + 23
nan
>>> c / 2
nan
>>> c * 2
nan
>>> math.sqrt(c)
nan
>>>
```

NaN值的一个特别的地方是它们之间的比较操作总是返回False。比如：

```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is d
False
>>>
```

由于这个原因，测试一个NaN值得唯一安全的方法就是使用 `math.isnan()`，也就是上面演示的那样。

有时候程序员想改变Python默认行为，在返回无穷大或NaN结果的操作中抛出异常。

`fpectl` 模块可以用来改变这种行为，但是它在标准的Python构建中并没有被启用，它是平台相关的，并且针对的是专家级程序员。可以参考在线的Python文档获取更多的细节。

3.8 分数运算

问题

你进入时间机器，突然发现你正在做小学家庭作业，并涉及到分数计算问题。或者你可能需要写代码去计算在你的木工工厂中的测量值。

解决方案

`fractions` 模块可以被用来执行包含分数的数学运算。比如：

```
>>> from fractions import Fraction
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b)
35/64

>>> # Getting numerator/denominator
>>> c = a * b
>>> c.numerator
35
>>> c.denominator
64

>>> # Converting to a float
>>> float(c)
0.546875

>>> # Limiting the denominator of a value
>>> print(c.limit_denominator(8))
4/7

>>> # Converting a float to a fraction
>>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(15, 4)
>>>
```

讨论

在大多数程序中一般不会出现分数的计算问题，但是有时候还是需要用到的。比如，在一个允许接受分数形式的测试单位并以分数形式执行运算的程序中，直接使用分数可以减少手动转换为小数或浮点数的工作。

3.9 大型数组运算

问题

你需要在大数据集(比如数组或网格)上面执行计算。

解决方案

涉及到数组的重量级运算操作，可以使用 `NumPy` 库。`NumPy` 的一个主要特征是它会给 Python 提供一个数组对象，相比标准的 Python 列表而已更适合用来做数学运算。下面是一个简单的小例子，向你展示标准列表对象和 `NumPy` 数组对象之间的差别：

```
>>> # Python Lists
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7, 8]
>>> x * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x + y
[1, 2, 3, 4, 5, 6, 7, 8]

>>> # Numpy arrays
>>> import numpy as np
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6,  8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])
>>>
```

正如所见，两种方案中数组的基本数学运算结果并不相同。特别的，`NumPy` 中的标量运算(比如 `ax * 2` 或 `ax + 10`)会作用在每一个元素上。另外，当两个操作数都是数组的时候执行元素对等位置计算，并最终生成一个新的数组。

对整个数组中所有元素同时执行数学运算可以使得作用在整个数组上的函数运算简单而又快速。比如，如果你想计算多项式的值，可以这样做：

```
>>> def f(x):
...     return 3*x**2 - 2*x + 7
...
>>> f(ax)
array([ 8, 15, 28, 47])
>>>
```

`NumPy` 还为数组操作提供了大量的通用函数，这些函数可以作为 `math` 模块中类似函数的替代。比如：

```
>>> np.sqrt(ax)
array([ 1. , 1.41421356, 1.73205081, 2. ])
>>> np.cos(ax)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
>>>
```

使用这些通用函数要比循环数组并使用 `math` 模块中的函数执行计算要快的多。因此，只要有可能的话尽量选择 `NumPy` 的数组方案。

底层实现中，`NumPy` 数组使用了C或者Fortran语言的机制分配内存。也就是说，它们是一个非常大的连续的并由同类型数据组成的内存区域。所以，你可以构造一个比普通Python列表大的多的数组。比如，如果你想构造一个10,000*10,000的浮点数二维网格，很轻松：

```
>>> grid = np.zeros(shape=(10000,10000), dtype=float)
>>> grid
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
>>>
```

所有的普通操作还是会同时作用在所有元素上：

```
>>> grid += 10
>>> grid
array([[ 10., 10., 10., ..., 10., 10., 10.],
       [ 10., 10., 10., ..., 10., 10., 10.],
       [ 10., 10., 10., ..., 10., 10., 10.],
       ...,
       [ 10., 10., 10., ..., 10., 10., 10.],
       [ 10., 10., 10., ..., 10., 10., 10.],
       [ 10., 10., 10., ..., 10., 10., 10.]])
>>> np.sin(grid)
array([[ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       ...,
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111]])
>>>
```

关于 `NumPy` 有一点需要特别的主意，那就是它扩展Python列表的索引功能 - 特别是对于多维数组。为了说明清楚，先构造一个简单的二维数组并试着做些试验：

```

>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a
array([[ 1, 2, 3, 4],
       [ 5, 6, 7, 8],
       [ 9, 10, 11, 12]])

>>> # Select row 1
>>> a[1]
array([5, 6, 7, 8])

>>> # Select column 1
>>> a[:,1]
array([ 2, 6, 10])

>>> # Select a subregion and change it
>>> a[1:3, 1:3]
array([[ 6, 7],
       [10, 11]])
>>> a[1:3, 1:3] += 10
>>> a
array([[ 1, 2, 3, 4],
       [ 5, 16, 17, 8],
       [ 9, 20, 21, 12]])

>>> # Broadcast a row vector across an operation on all rows
>>> a + [100, 101, 102, 103]
array([[101, 103, 105, 107],
       [105, 117, 119, 111],
       [109, 121, 123, 115]])
>>> a
array([[ 1, 2, 3, 4],
       [ 5, 16, 17, 8],
       [ 9, 20, 21, 12]])

>>> # Conditional assignment on an array
>>> np.where(a < 10, a, 10)
array([[ 1, 2, 3, 4],
       [ 5, 10, 10, 8],
       [ 9, 10, 10, 10]])
>>>

```

讨论

`NumPy` 是Python领域中很多科学与工程库的基础，同时也是被广泛使用的最大最复杂的模块。即便如此，在刚开始的时候通过一些简单的例子和玩具程序也能帮我们完成一些有趣的事情。

通常我们导入 `NumPy` 模块的时候会使用语句 `import numpy as np`。这样的话你就不用再你的程序里面一遍遍的敲入 `numpy`，只需要输入 `np` 就行了，节省了不少时间。

如果想获取更多的信息，你当然得去 `NumPy` 官网逛逛了，网址是：

<http://www.numpy.org>

3.10 矩阵与线性代数运算

问题

你需要执行矩阵和线性代数运算，比如矩阵乘法、寻找行列式、求解线性方程组等等。

解决方案

`NumPy` 库有一个矩阵对象可以用来解决这个问题。

矩阵类似于3.9小节中数组对象，但是遵循线性代数的计算规则。下面的一个例子展示了矩阵的一些基本特性：

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2, 3],
        [ 0, 4, 5],
        [ 7, 8, -9]])

>>> # Return transpose
>>> m.T
matrix([[ 1, 0, 7],
        [-2, 4, 8],
        [ 3, 5, -9]])

>>> # Return inverse
>>> m.I
matrix([[ 0.33043478, -0.02608696, 0.09565217],
        [-0.15217391, 0.13043478, 0.02173913],
        [ 0.12173913, 0.09565217, -0.0173913 ]])

>>> # Create a vector and multiply
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],
        [3],
        [4]])
>>> m * v
matrix([[ 8],
        [32],
        [ 2]])
>>>
```

可以在 `numpy.linalg` 子包中找到更多的操作函数，比如：

```
>>> import numpy.linalg

>>> # Determinant
>>> numpy.linalg.det(m)
-229.99999999999983

>>> # Eigenvalues
>>> numpy.linalg.eigvals(m)
array([-13.11474312,  2.75956154,  6.35518158])

>>> # Solve for x in mx = v
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
>>> m * x
matrix([[ 2.],
        [ 3.],
        [ 4.]])
>>> v
matrix([[2],
        [3],
        [4]])
>>>
```

讨论

很显然线性代数是个非常大的主题，已经超出了本书能讨论的范围。但是，如果你需要操作数组和向量的话，`NumPy` 是一个不错的入口点。可以访问 `NumPy` 官网 <http://www.numpy.org> 获取更多信息。

3.11 随机选择

问题

你想从一个序列中随机抽取若干元素，或者想生成几个随机数。

解决方案

`random` 模块有大量的函数用来产生随机数和随机选择元素。比如，要想从一个序列中随机的抽取一个元素，可以使用 `random.choice()`：

```
>>> import random
>>> values = [1, 2, 3, 4, 5, 6]
>>> random.choice(values)
2
>>> random.choice(values)
3
>>> random.choice(values)
1
>>> random.choice(values)
4
>>> random.choice(values)
6
>>>
```

为了提取出N个不同元素的样本用来做进一步的操作，可以使用 `random.sample()`：

```
>>> random.sample(values, 2)
[6, 2]
>>> random.sample(values, 2)
[4, 3]
>>> random.sample(values, 3)
[4, 3, 1]
>>> random.sample(values, 3)
[5, 4, 1]
>>>
```

如果你仅仅只是想打乱序列中元素的顺序，可以使用 `random.shuffle()`：

```
>>> random.shuffle(values)
>>> values
[2, 4, 6, 5, 3, 1]
>>> random.shuffle(values)
>>> values
[3, 5, 2, 1, 6, 4]
>>>
```

生成随机整数，请使用 `random.randint()`：

```
>>> random.randint(0,10)
2
>>> random.randint(0,10)
5
>>> random.randint(0,10)
0
>>> random.randint(0,10)
7
>>> random.randint(0,10)
10
>>> random.randint(0,10)
3
>>>
```

为了生成0到1范围内均匀分布的浮点数，使用 `random.random()`：

```
>>> random.random()
0.9406677561675867
>>> random.random()
0.133129581343897
>>> random.random()
0.4144991136919316
>>>
```

如果要获取N位随机位(二进制)的整数，使用 `random.getrandbits()`：

```
>>> random.getrandbits(200)
335837000776573622800628485064121869519521710558559406913275
>>>
```

讨论

`random` 模块使用 Mersenne Twister 算法来计算生成随机数。这是一个确定性算法，但是您可以通过 `random.seed()` 函数修改初始化种子。比如：

```
random.seed() # Seed based on system time or os.urandom()
random.seed(12345) # Seed based on integer given
random.seed(b'bytedata') # Seed based on byte data
```

除了上述介绍的功能，`random`模块还包含基于均匀分布、高斯分布和其他分布的随机数生成函数。比如，`random.uniform()` 计算均匀分布随机数，`random.gauss()` 计算正态分布随机数。对于其他的分布情况请参考在线文档。

在 `random` 模块中的函数不应该用在和密码学相关的程序中。如果你确实需要类似的功能，可以使用`ssl`模块中相应的函数。比如，`ssl.RAND_bytes()` 可以用来生成一个安全的随机字节序列。

3.12 基本的日期与时间转换

问题

你需要执行简单的时间转换，比如天到秒，小时到分钟等的转换。

解决方案

为了执行不同时间单位的转换和计算，请使用 `datetime` 模块。比如，为了表示一个时间段，可以创建一个 `timedelta` 实例，就像下面这样：

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

如果你想表示指定的日期和时间，先创建一个 `datetime` 实例然后使用标准的数学运算来操作它们。比如：

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
>>>
```

在计算的时候，需要注意的是 `datetime` 会自动处理闰年。比如：

```
>>> a = datetime(2012, 3, 1)
>>> b = datetime(2012, 2, 28)
>>> a - b
datetime.timedelta(2)
>>> (a - b).days
2
>>> c = datetime(2013, 3, 1)
>>> d = datetime(2013, 2, 28)
>>> (c - d).days
1
>>>
```

讨论

对大多数基本的日期和时间处理问题，`datetime` 模块以及足够了。如果你需要执行更加复杂的日期操作，比如处理时区，模糊时间范围，节假日计算等等，可以考虑使用 [dateutil](#) 模块

许多类似的时间计算可以使用 `dateutil.relativedelta()` 函数代替。但是，有一点需要注意的就是，它会在处理月份(还有它们的天数差距)的时候填充间隙。看例子最清楚：

```
>>> a = datetime(2012, 9, 23)
>>> a + timedelta(months=1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'months' is an invalid keyword argument for this function
>>>
>>> from dateutil.relativedelta import relativedelta
>>> a + relativedelta(months=+1)
datetime.datetime(2012, 10, 23, 0, 0)
>>> a + relativedelta(months=+4)
datetime.datetime(2013, 1, 23, 0, 0)
>>>
>>> # Time between two dates
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d
datetime.timedelta(89)
>>> d = relativedelta(b, a)
>>> d
relativedelta(months=+2, days=+28)
>>> d.months
2
>>> d.days
28
>>>
```

3.13 计算最后一个周五的日期

问题

你需要查找星期中某一天最后出现的日期，比如星期五。

解决方案

Python的 `datetime` 模块中有工具函数和类可以帮助你执行这样的计算。下面是对类似这样的问题的一个通用解决方案：

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 最后的周五
Desc :
"""
from datetime import datetime, timedelta

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
             'Friday', 'Saturday', 'Sunday']

def get_previous_byday(dayname, start_date=None):
    if start_date is None:
        start_date = datetime.today()
    day_num = start_date.weekday()
    day_num_target = weekdays.index(dayname)
    days_ago = (7 + day_num - day_num_target) % 7
    if days_ago == 0:
        days_ago = 7
    target_date = start_date - timedelta(days=days_ago)
    return target_date
```

在交互式解释器中使用如下：

```
>>> datetime.today() # For reference
datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
>>> get_previous_byday('Monday')
datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
>>> get_previous_byday('Tuesday') # Previous week, not today
datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
>>> get_previous_byday('Friday')
datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
>>>
```

可选的 `start_date` 参数可以由另外一个 `datetime` 实例来提供。比如：

```
>>> get_previous_byday('Sunday', datetime(2012, 12, 21))
datetime.datetime(2012, 12, 16, 0, 0)
>>>
```

讨论

上面的算法原理是这样的：先将开始日期和目标日期映射到星期数组的位置上(星期一索引为0)，然后通过模运算计算出目标日期要经过多少天才能到达开始日期。然后用开始日期减去那个时间差即得到结果日期。

如果你要像这样执行大量的日期计算的话，你最好安装第三方包 `python-dateutil` 来代替。比如，下面是使用 `dateutil` 模块中的 `relativedelta()` 函数执行同样的计算：

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
>>> print(d)
2012-12-23 16:31:52.718111

>>> # Next Friday
>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>

>>> # Last Friday
>>> print(d + relativedelta(weekday=FR(-1)))
2012-12-21 16:31:52.718111
>>>
```

3.14 计算当前月份的日期范围

问题

你的代码需要在当前月份中循环每一天，想找到一个计算这个日期范围的高效方法。

解决方案

在这样的日期上循环并需要事先构造一个包含所有日期的列表。你可以先计算出开始日期和结束日期，然后在你步进的时候使用 `datetime.timedelta` 对象递增这个日期变量即可。

下面是一个接受任意 `datetime` 对象并返回一个由当前月份开始日和下个月开始日组成的元组对象。

```
from datetime import datetime, date, timedelta
import calendar

def get_month_range(start_date=None):
    if start_date is None:
        start_date = date.today().replace(day=1)
    _, days_in_month = calendar.monthrange(start_date.year, start_date.month)
    end_date = start_date + timedelta(days=days_in_month)
    return (start_date, end_date)
```

有了这个就可以很容易的在返回的日期范围上面做循环操作了：

```
>>> a_day = timedelta(days=1)
>>> first_day, last_day = get_month_range()
>>> while first_day < last_day:
...     print(first_day)
...     first_day += a_day
...
2012-08-01
2012-08-02
2012-08-03
2012-08-04
2012-08-05
2012-08-06
2012-08-07
2012-08-08
2012-08-09
#... and so on...
```

讨论

上面的代码先计算出一个对应月份第一天的日期。一个快速的方法就是使用 `date` 或 `datetime` 对象的 `replace()` 方法简单的将 `days` 属性设置成1即可。 `replace()` 方法一个好处就是它会创建和你开始传入对象类型相同的对象。所以，如果输入参数是一个 `date` 实例，那么结果也是一个 `date` 实例。同样的，如果输入是一个 `datetime` 实例，那么你得到的就是一个 `datetime` 实例。

然后，使用 `calendar.monthrange()` 函数来找出该月的总天数。任何时候只要你想获得日历信息，那么 `calendar` 模块就非常有用。 `monthrange()` 函数会返回包含星期和该月天数的元组。

一旦该月的天数已知了，那么结束日期就可以通过在开始日期上面加上这个天数获得。有个需要注意的是结束日期并不包含在这个日期范围内(事实上它是下个月的开始日期)。这个和Python的 `slice` 与 `range` 操作行为保持一致，同样也不包含结尾。

为了在日期范围上循环，要使用到标准的数学和比较操作。比如，可以利用 `timedelta` 实例来递增日期，小于号<用来检查一个日期是否在结束日期之前。

理想情况下，如果能为日期迭代创建一个同内置的 `range()` 函数一样的函数就好了。幸运的是，可以使用一个生成器来很容易的实现这个目标：

```
def date_range(start, stop, step):
    while start < stop:
        yield start
        start += step
```

下面是使用这个生成器的例子：

```
>>> for d in date_range(datetime(2012, 9, 1), datetime(2012,10,1),
...                       timedelta(hours=6)):
...     print(d)
...
2012-09-01 00:00:00
2012-09-01 06:00:00
2012-09-01 12:00:00
2012-09-01 18:00:00
2012-09-02 00:00:00
2012-09-02 06:00:00
...
>>>
```

这种实现之所以这么简单，还得归功于Python中的日期和时间能够使用标准的数学和比较操作符来进行运算。

3.15 字符串转换为日期

问题

你的应用程序接受字符串格式的输入，但是你想将它们转换为 `datetime` 对象以便在上面执行非字符串操作。

解决方案

使用Python的标准模块 `datetime` 可以很容易的解决这个问题。比如：

```
>>> from datetime import datetime
>>> text = '2012-09-20'
>>> y = datetime.strptime(text, '%Y-%m-%d')
>>> z = datetime.now()
>>> diff = z - y
>>> diff
datetime.timedelta(3, 77824, 177393)
>>>
```

讨论

`datetime.strptime()` 方法支持很多的格式化代码，比如 `%Y` 代表4位数年份，`%m` 代表两位数月份。还有一点值得注意的是这些格式化占位符也可以反过来使用，将日期输出为指定的格式字符串形式。

比如，假设你的代码中生成了一个 `datetime` 对象，你想将它格式化为漂亮易读形式后放在自动生成的信件或者报告的顶部：

```
>>> z
datetime.datetime(2012, 9, 23, 21, 37, 4, 177393)
>>> nice_z = datetime.strftime(z, '%A %B %d, %Y')
>>> nice_z
'Sunday September 23, 2012'
>>>
```

还有一点需要注意的是，`strptime()` 的性能要比你想象中的差很多，因为它是使用纯Python实现，并且必须处理所有的系统本地设置。如果你要在代码中需要解析大量的日期并且已经知道了日期字符串的确切格式，可以自己实现一套解析方案来获取更好的性能。比如，如果你已经知道所以日期格式是 `YYYY-MM-DD`，你可以像下面这样实现一个解析函数：

```
from datetime import datetime
def parse_ymd(s):
    year_s, mon_s, day_s = s.split('-')
    return datetime(int(year_s), int(mon_s), int(day_s))
```

实际测试中，这个函数比 `datetime.strptime()` 快7倍多。如果你要处理大量的涉及到日期的数据的话，那么最好考虑下这个方案！

3.16 结合时区的日期操作

问题

你有一个安排在2012年12月21日早上9:30的电话会议，地点在芝加哥。而你的朋友在印度的班加罗尔，那么他应该在当地时间几点参加这个会议呢？

解决方案

对几乎所有涉及到时区的问题，你都应该使用 `pytz` 模块。这个包提供了Olson时区数据库，它是时区信息的事实上的标准，在很多语言和操作系统里面都可以找到。

`pytz` 模块一个主要用途是将 `datetime` 库创建的简单日期对象本地化。比如，下面如何表示一个芝加哥时间的示例：

```
>>> from datetime import datetime
>>> from pytz import timezone
>>> d = datetime(2012, 12, 21, 9, 30, 0)
>>> print(d)
2012-12-21 09:30:00
>>>

>>> # Localize the date for Chicago
>>> central = timezone('US/Central')
>>> loc_d = central.localize(d)
>>> print(loc_d)
2012-12-21 09:30:00-06:00
>>>
```

一旦日期被本地化了，它就可以转换为其他时区的时间了。为了得到班加罗尔对应的时间，你可以这样做：

```
>>> # Convert to Bangalore time
>>> bang_d = loc_d.astimezone(timezone('Asia/Kolkata'))
>>> print(bang_d)
2012-12-21 21:00:00+05:30
>>>
```

如果你打算在本地化日期上执行计算，你需要特别注意夏令时转换和其他细节。比如，在2013年，美国标准夏令时时间开始于本地时间3月13日凌晨2:00(在那时，时间向前跳过一小时)。如果你正在执行本地计算，你会得到一个错误。比如：

```
>>> d = datetime(2013, 3, 10, 1, 45)
>>> loc_d = central.localize(d)
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> later = loc_d + timedelta(minutes=30)
>>> print(later)
2013-03-10 02:15:00-06:00 # WRONG! WRONG!
>>>
```

结果错误是因为它并没有考虑在本地时间中有一小时的跳跃。为了修正这个错误，可以使用时区对象 `normalize()` 方法。比如：

```
>>> from datetime import timedelta
>>> later = central.normalize(loc_d + timedelta(minutes=30))
>>> print(later)
2013-03-10 03:15:00-05:00
>>>
```

讨论

为了不让你被这些东东弄的晕头转向，处理本地化日期的通常的策略先将所有日期转换为UTC时间，并用它来执行所有的中间存储和操作。比如：

```
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> utc_d = loc_d.astimezone(pytz.utc)
>>> print(utc_d)
2013-03-10 07:45:00+00:00
>>>
```

一旦转换为UTC，你就不用去担心跟夏令时相关的问题了。因此，你可以跟之前一样放心的执行常见的日期计算。当你想将输出变为本地时间的时候，使用合适的时区去转换下就行了。比如：

```
>>> later_utc = utc_d + timedelta(minutes=30)
>>> print(later_utc.astimezone(central))
2013-03-10 03:15:00-05:00
>>>
```

当涉及到时区操作的时候，有个问题就是我们如何得到时区的名称。比如，在这个例子中，我们如何知道“Asia/Kolkata”就是印度对应的时区名呢？为了查找，可以使用ISO 3166国家代码作为关键字去查阅字典 `pytz.country_timezones`。比如：

```
>>> pytz.country_timezones['IN']
['Asia/Kolkata']
>>>
```

注：当你阅读到这里的时候，有可能 `pytz` 模块以及不再建议使用，因为PEP431提出了更先进的时区支持。但是这里谈到的很多问题还是有参考价值的(比如使用UTC日期的建议等)。

第四章：迭代器与生成器

迭代是Python最强大的功能之一。初看起来，你可能会简单的认为迭代只不过是处理序列中元素的一种方法。然而，绝非仅仅就是如此，还有很多你可能不知道的，比如创建你自己的迭代器对象，在itertools模块中使用有用的迭代模式，构造生成器函数等等。这一章目的就是向你展示跟迭代有关的各种常见问题。

Contents:

4.1 手动遍历迭代器

问题

你想遍历一个可迭代对象中的所有元素，但是却不想使用for循环。

解决方案

为了手动的遍历可迭代对象，使用 `next()` 函数并在代码中捕获 `StopIteration` 异常。比如，下面的例子手动读取一个文件中的所有行：

```
def manual_iter():
    with open('/etc/passwd') as f:
        try:
            while True:
                line = next(f)
                print(line, end='')
        except StopIteration:
            pass
```

通常来讲，`StopIteration` 用来指示迭代的结尾。然而，如果你手动使用上面演示的 `next()` 函数的话，你还可以通过返回一个指定值来标记结尾，比如 `None`。下面是示例：

```
with open('/etc/passwd') as f:
    while True:
        line = next(f)
        if line is None:
            break
        print(line, end='')
```

讨论

大多数情况下，我们会使用 `for` 循环语句用来遍历一个可迭代对象。但是，偶尔也需要对迭代做更加精确的控制，这时候了解底层迭代机制就显得尤为重要了。

下面的交互示例向我们演示了迭代期间所发生的基本细节：

```
>>> items = [1, 2, 3]
>>> # Get the iterator
>>> it = iter(items) # Invokes items.__iter__()
>>> # Run the iterator
>>> next(it) # Invokes it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

本章接下来几小节会更深入的讲解迭代相关技术，前提是你先要理解基本的迭代协议机制。所以确保你已经把这章的内容牢牢记在心中。

4.2 代理迭代

问题

你构建了一个自定义容器对象，里面包含有列表、元组或其他可迭代对象。你想直接在你的这个新容器对象上执行迭代操作。

解决方案

实际上你只需要定义一个 `__iter__()` 方法，将迭代操作代理到容器内部的对象上去。比如：

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    # Outputs Node(1), Node(2)
    for ch in root:
        print(ch)
```

在上面代码中，`__iter__()` 方法只是简单的将迭代请求传递给内部的 `_children` 属性。

讨论

Python的迭代器协议需要 `__iter__()` 方法返回一个实现了 `__next__()` 方法的迭代器对象。如果你只是迭代遍历其他容器的内容，你无须担心底层是怎样实现的。你所要做的只是传递迭代请求既可。

这里的 `iter()` 函数的使用简化了代码，`iter(s)` 只是简单的通过调用 `s.__iter__()` 方法来返回对应的迭代器对象，就跟 `len(s)` 会调用 `s.__len__()` 原理是一样的。

4.3 使用生成器创建新的迭代模式

问题

你想实现一个自定义迭代模式，跟普通的内置函数比如 `range()`，`reversed()` 不一样。

解决方案

如果你想实现一种新的迭代模式，使用一个生成器函数来定义它。下面是一个生产某个范围内浮点数的生成器：

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment
```

为了使用这个函数，你可以用 `for` 循环迭代它或者使用其他接受一个可迭代对象的函数(比如 `sum()`，`list()` 等)。示例如下：

```
>>> for n in frange(0, 4, 0.5):
...     print(n)
...
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

讨论

一个函数中需要有一个 `yield` 语句即可将其转换为一个生成器。跟普通函数不同的是，生成器只能用于迭代操作。下面是一个实验，向你展示这样的函数底层工作机制：

```

>>> def countdown(n):
...     print('Starting to count from', n)
...     while n > 0:
...         yield n
...         n -= 1
...     print('Done!')
...

>>> # Create the generator, notice no output appears
>>> c = countdown(3)
>>> c
<generator object countdown at 0x1006a0af0>

>>> # Run to first yield and emit a value
>>> next(c)
Starting to count from 3
3

>>> # Run to the next yield
>>> next(c)
2

>>> # Run to next yield
>>> next(c)
1

>>> # Run to next yield (iteration stops)
>>> next(c)
Done!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>

```

一个生成器函数主要特征是它只会回应在使用到的 `next` 操作。一旦生成器函数返回退出，迭代终止。我们在迭代中通常使用的 `for` 语句会自动处理这些细节，所以你无需担心。

4.4 实现迭代器协议

问题

你想构建一个能支持迭代操作的自定义对象，并希望找到一个能实现迭代协议的简单方法。

解决方案

目前为止，在一个对象上实现迭代最简单的方式是使用一个生成器函数。在4.2小节中，使用Node类来表示树形数据结构。你可能想实现一个以深度优先方式遍历树形节点的生成器。下面是代码示例：

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        yield self
        for c in self:
            yield from c.depth_first()

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch)
# Outputs Node(0), Node(1), Node(3), Node(4), Node(2), Node(5)
```

在这段代码中，`depth_first()` 方法简单直观。它首先返回自己本身并迭代每一个子节点并通过调用子节点的 `depth_first()` 方法(使用 `yield from` 语句)返回对应元素。

讨论

Python的迭代协议要求一个 `__iter__()` 方法返回一个特殊的迭代器对象，这个迭代器对象实现了 `__next__()` 方法并通过 `StopIteration` 异常标识迭代的完成。但是，实现这些通常会比较繁琐。下面我们演示下这种方式，如何使用一个关联迭代器类重新实现 `depth_first()` 方法：

```

class Node2:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        return DepthFirstIterator(self)

class DepthFirstIterator(object):
    """
    Depth-first traversal
    """

    def __init__(self, start_node):
        self._node = start_node
        self._children_iter = None
        self._child_iter = None

    def __iter__(self):
        return self

    def __next__(self):
        # Return myself if just started; create an iterator for children
        if self._children_iter is None:
            self._children_iter = iter(self._node)
            return self._node
        # If processing a child, return its next item
        elif self._child_iter:
            try:
                nextchild = next(self._child_iter)
                return nextchild
            except StopIteration:
                self._child_iter = None
                return next(self)
        # Advance to the next child and start its iteration
        else:
            self._child_iter = next(self._children_iter).depth_first()
            return next(self)

```

`DepthFirstIterator` 类和上面使用生成器的版本工作原理类似，但是它写起来很繁琐，因为迭代器必须在迭代处理过程中维护大量的状态信息。坦白来讲，没人愿意写这么晦涩的代码。将你的迭代器定义为一个生成器后一切迎刃而解。

4.5 反向迭代

问题

你想反方向迭代一个序列

解决方案

使用内置的 `reversed()` 函数，比如：

```
>>> a = [1, 2, 3, 4]
>>> for x in reversed(a):
...     print(x)
...
4
3
2
1
```

反向迭代仅仅当对象的大小可预先确定或者对象实现了 `__reversed__()` 的特殊方法时才能生效。如果两者都不符合，那你必须先将对象转换为一个列表才行，比如：

```
# Print a file backwards
f = open('somefile')
for line in reversed(list(f)):
    print(line, end='')
```

要注意的是如果可迭代对象元素很多的话，将其预先转换为一个列表要消耗大量的内存。

讨论

很多程序员并不知道可以通过在自定义类上实现 `__reversed__()` 方法来实现反向迭代。比如：

```
class Countdown:
    def __init__(self, start):
        self.start = start

    # Forward iterator
    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1

    # Reverse iterator
    def __reversed__(self):
        n = 1
        while n <= self.start:
            yield n
            n += 1

for rr in reversed(Countdown(30)):
    print(rr)
for rr in Countdown(30):
    print(rr)
```

定义一个反向迭代器可以使得代码非常的高效，因为它不再需要将数据填充到一个列表中然后再去反向迭代这个列表。

4.6 带有外部状态的生成器函数

问题

你想定义一个生成器函数，但是它会调用某个你想暴露给用户使用的外部状态值。

解决方案

如果你想让你的生成器暴露外部状态给用户，别忘了你可以简单的将它实现为一个类，然后把生成器函数放到 `__iter__()` 方法中过去。比如：

```
from collections import deque

class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines = lines
        self.history = deque(maxlen=histlen)

    def __iter__(self):
        for lineno, line in enumerate(self.lines, 1):
            self.history.append((lineno, line))
            yield line

    def clear(self):
        self.history.clear()
```

为了使用这个类，你可以将它当做是一个普通的生成器函数。然而，由于可以创建一个实例对象，于是你可以访问内部属性值，比如 `history` 属性或者是 `clear()` 方法。代码示例如下：

```
with open('somefile.txt') as f:
    lines = linehistory(f)
    for line in lines:
        if 'python' in line:
            for lineno, hline in lines.history:
                print('{}:{}'.format(lineno, hline), end='')
```

讨论

关于生成器，很容易掉进函数无所不能的陷阱。如果生成器函数需要跟你的程序其他部分打交道的話(比如暴露属性值，允许通过方法调用来控制等等)，可能会导致你的代码异常的复杂。如果是这种情况的话，可以考虑使用上面介绍的定义类的方式。在 `__iter__()` 方法中定义你的生成器不会改变你任何的算法逻辑。由于它是类的一部分，所以允许你定义各种属性和方法来供用户使用。

一个需要注意的小地方是，如果你在迭代操作时不使用 `for` 循环语句，那么你得先调用 `iter()` 函数。比如：


```
>>> f = open('somefile.txt')
>>> lines = linehistory(f)
>>> next(lines)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'linehistory' object is not an iterator

>>> # Call iter() first, then start iterating
>>> it = iter(lines)
>>> next(it)
'hello world\n'
>>> next(it)
'this is a test\n'
>>>
```

4.7 迭代器切片

问题

你想得到一个由迭代器生成的切片对象，但是标准切片操作并不能做到。

解决方案

函数 `itertools.islice()` 正好适用于在迭代器和生成器上做切片操作。比如：

```

>>> def count(n):
...     while True:
...         yield n
...         n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable

>>> # Now using islice()
>>> import itertools
>>> for x in itertools.islice(c, 10, 20):
...     print(x)
...
10
11
12
13
14
15
16
17
18
19
>>>

```

讨论

迭代器和生成器不能使用标准的切片操作，因为它们的长度事先我们并不知道(并且也没有实现索引)。函数 `islice()` 返回一个可以生成指定元素的迭代器，它通过遍历并丢弃直到切片开始索引位置的所有元素。然后才开始一个个的返回元素，并直到切片结束索引位置。

这里要着重强调的一点是 `islice()` 会消耗掉传入的迭代器中的数据。必须考虑到迭代器是不可逆的这个事实。所以如果你需要之后再次访问这个迭代器的话，那你就得先将它里面的数据放入一个列表中。

4.8 跳过可迭代对象的开始部分

问题

你想遍历一个可迭代对象，但是它开始的某些元素你并不感兴趣，想跳过它们。

解决方案

`itertools` 模块中有一些函数可以完成这个任务。首先介绍的是 `itertools.dropwhile()` 函数。使用时，你给它传递一个函数对象和一个可迭代对象。它会返回一个迭代器对象，丢弃原有序列中直到函数返回`True`之前的所有元素，然后返回后面所有元素。

为了演示，假定你在读取一个开始部分是几行注释的源文件。比如：

```
>>> with open('/etc/passwd') as f:
...     for line in f:
...         print(line, end='')
...
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode. At other times, this information is provided by
# Open Directory.
...
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

如果你想跳过开始部分的注释行的话，可以这样做：

```
>>> from itertools import dropwhile
>>> with open('/etc/passwd') as f:
...     for line in dropwhile(lambda line: line.startswith('#'), f):
...         print(line, end='')
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

这个例子是基于根据某个测试函数跳过开始的元素。如果你已经明确知道了要跳过的元素的个数的话，那么可以使用 `itertools.islice()` 来代替。比如：

```
>>> from itertools import islice
>>> items = ['a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
...     print(x)
...
1
4
10
15
>>>
```

在这个例子中，`islice()` 函数最后那个 `None` 参数指定了你要获取从第3个到最后的所有元素，如果 `None` 和3的位置对调，意思就是仅仅获取前三个元素恰恰相反，（这个跟切片的相反操作 `[3:]` 和 `[:3]` 原理是一样的）。

讨论

函数 `dropwhile()` 和 `islice()` 其实就是两个帮助函数，为的就是避免写出下面这种冗余代码：

```
with open('/etc/passwd') as f:
    # Skip over initial comments
    while True:
        line = next(f, '')
        if not line.startswith('#'):
            break

    # Process remaining lines
    while line:
        # Replace with useful processing
        print(line, end='')
        line = next(f, None)
```

跳过一个可迭代对象的开始部分跟通常的过滤是不同的。比如，上述代码的第一个部分可能会这样重写：

```
with open('/etc/passwd') as f:
    lines = (line for line in f if not line.startswith('#'))
    for line in lines:
        print(line, end='')
```

这样写确实可以跳过开始部分的注释行，但是同样也会跳过文件中其他所有的注释行。换句话说，我们的解决方案是仅仅跳过开始部分满足测试条件的行，在那以后，所有的元素不再进行测试和过滤了。

最后需要着重强调的一点是，本节的方案适用于所有可迭代对象，包括那些事先不能确定大小的，比如生成器，文件及其类似的对象。

4.9 排列组合的迭代

问题

你想迭代遍历一个集合中元素的所有可能的排列或组合

解决方案

`itertools`模块提供了三个函数来解决这类问题。其中一个是 `itertools.permutations()`，它接受一个集合并产生一个元组序列，每个元组由集合中所有元素的一个可能排列组成。也就是说通过打乱集合中元素排列顺序生成一个元组，比如：

```
>>> items = ['a', 'b', 'c']
>>> from itertools import permutations
>>> for p in permutations(items):
...     print(p)
...
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
```

如果你想得到指定长度的所有排列，你可以传递一个可选的长度参数。就像这样：

```
>>> for p in permutations(items, 2):
...     print(p)
...
('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
>>>
```

使用 `itertools.combinations()` 可得到输入集合中元素的所有组合。比如：

```
>>> from itertools import combinations
>>> for c in combinations(items, 3):
...     print(c)
...
('a', 'b', 'c')

>>> for c in combinations(items, 2):
...     print(c)
...
('a', 'b')
('a', 'c')
('b', 'c')

>>> for c in combinations(items, 1):
...     print(c)
...
('a',)
('b',)
('c',)
>>>
```

对于 `combinations()` 来讲，元素的顺序已经不重要了。也就是说，组合 `('a', 'b')` 跟 `('b', 'a')` 其实是一样的(最终只会输出其中一个)。

在计算组合的时候，一旦元素被选取就会从候选中剔除掉(比如如果元素'a'已经被选取了，那么接下来就不会再考虑它了)。而函数 `itertools.combinations_with_replacement()` 允许同一个元素被选择多次，比如：

```
>>> for c in combinations_with_replacement(items, 3):
...     print(c)
...
('a', 'a', 'a')
('a', 'a', 'b')
('a', 'a', 'c')
('a', 'b', 'b')
('a', 'b', 'c')
('a', 'c', 'c')
('b', 'b', 'b')
('b', 'b', 'c')
('b', 'c', 'c')
('c', 'c', 'c')
>>>
```

讨论

这一小节我们向你展示的仅仅是 `itertools` 模块的一部分功能。尽管你也可以自己手动实现排列组合算法，但是这样做得要花点脑力。当我们碰到看上去有些复杂的迭代问题时，最好可以先去看看 `itertools` 模块。如果这个问题很普遍，那么很有可能会在里面找到解决方案！

4.10 序列上索引值迭代

问题

你想在迭代一个序列的同时跟踪正在被处理的元素索引。

解决方案

内置的 `enumerate()` 函数可以很好的解决这个问题：

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list):
...     print(idx, val)
...
0 a
1 b
2 c
```

为了按传统行号输出(行号从1开始)，你可以传递一个开始参数：

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list, 1):
...     print(idx, val)
...
1 a
2 b
3 c
```

这种情况在你遍历文件时想在错误消息中使用行号定位时候非常有用：

```
def parse_data(filename):
    with open(filename, 'rt') as f:
        for lineno, line in enumerate(f, 1):
            fields = line.split()
            try:
                count = int(fields[1])
                ...
            except ValueError as e:
                print('Line {}: Parse error: {}'.format(lineno, e))
```

`enumerate()` 对于跟踪某些值在列表中出现的位置是很有用的。所以，如果你想将一个文件中出现的单词映射到它出现的行号上去，可以很容易的利用 `enumerate()` 来完成：

```
word_summary = defaultdict(list)

with open('myfile.txt', 'r') as f:
    lines = f.readlines()

for idx, line in enumerate(lines):
    # Create a List of words in current line
    words = [w.strip().lower() for w in line.split()]
    for word in words:
        word_summary[word].append(idx)
```

如果你处理完文件后打印 `word_summary`，会发现它是一个字典(准确来讲是一个 `defaultdict`)，对于每个单词有一个 `key`，每个 `key` 对应的值是一个由这个单词出现的行号组成的列表。如果某个单词在一行中出现过两次，那么这个行号也会出现两次，同时也可以作为文本的一个简单统计。

讨论

当你想额外定义一个计数变量的时候，使用 `enumerate()` 函数会更加简单。你可能会像下面这样写代码：

```
lineno = 1
for line in f:
    # Process line
    ...
    lineno += 1
```

但是如果使用 `enumerate()` 函数来代替就显得更加优雅了：


```
for lineno, line in enumerate(f):
    # Process line
    ...
```

`enumerate()` 函数返回的是一个 `enumerate` 对象实例，它是一个迭代器，返回连续的包含一个计数和一个值的元组，元组中的值通过在传入序列上调用 `next()` 返回。

还有一点可能并不很重要，但是也值得注意，有时候当你在一个已经解压后的元组序列上使用 `enumerate()` 函数时很容易调入陷阱。你得像下面正确的方式这样写：

```
data = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

# Correct!
for n, (x, y) in enumerate(data):
    ...
# Error!
for n, x, y in enumerate(data):
    ...
```

4.11 同时迭代多个序列

问题

你想同时迭代多个序列，每次分别从一个序列中取一个元素。

解决方案

为了同时迭代多个序列，使用 `zip()` 函数。比如：

```
>>> xpts = [1, 5, 4, 2, 10, 7]
>>> ypts = [101, 78, 37, 15, 62, 99]
>>> for x, y in zip(xpts, ypts):
...     print(x,y)
...
1 101
5 78
4 37
2 15
10 62
7 99
>>>
```

`zip(a, b)` 会生成一个可返回元组 `(x, y)` 的迭代器，其中 `x` 来自 `a`，`y` 来自 `b`。一旦其中某个序列到底结尾，迭代宣告结束。因此迭代长度跟参数中最短序列长度一致。

```
>>> a = [1, 2, 3]
>>> b = ['w', 'x', 'y', 'z']
>>> for i in zip(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
>>>
```

如果这个不是你想要的效果，那么还可以使用 `itertools.zip_longest()` 函数来代替。比如：

```
>>> from itertools import zip_longest
>>> for i in zip_longest(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(None, 'z')

>>> for i in zip_longest(a, b, fillvalue=0):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(0, 'z')
>>>
```

讨论

当你想成对处理数据的时候 `zip()` 函数是很有用的。比如，假设你头列表和一个值列表，就像下面这样：

```
headers = ['name', 'shares', 'price']
values = ['ACME', 100, 490.1]
```

使用`zip()`可以让你将它们打包并生成一个字典：

```
s = dict(zip(headers, values))
```

或者你也可以像下面这样产生输出：

```
for name, val in zip(headers, values):  
    print(name, '=', val)
```

虽然不常见，但是 `zip()` 可以接受多于两个的序列的参数。这时候所生成的结果元组中元素个数跟输入序列个数一样。比如：

```
>>> a = [1, 2, 3]  
>>> b = [10, 11, 12]  
>>> c = ['x', 'y', 'z']  
>>> for i in zip(a, b, c):  
...     print(i)  
...  
(1, 10, 'x')  
(2, 11, 'y')  
(3, 12, 'z')  
>>>
```

最后强调一点就是，`zip()` 会创建一个迭代器来作为结果返回。如果你需要将结对的值存储在列表中，要使用 `list()` 函数。比如：

```
>>> zip(a, b)  
<zip object at 0x1007001b8>  
>>> list(zip(a, b))  
[(1, 10), (2, 11), (3, 12)]  
>>>
```

4.12 不同集合上元素的迭代

问题

你想在多个对象执行相同的操作，但是这些对象在不同的容器中，你希望代码在不失可读性的情况下避免写重复的循环。

解决方案

`itertools.chain()` 方法可以用来简化这个任务。它接受一个可迭代对象列表作为输入，并返回一个迭代器，有效的屏蔽掉在多个容器中迭代细节。为了演示清楚，考虑下面这个例子：

```
>>> from itertools import chain
>>> a = [1, 2, 3, 4]
>>> b = ['x', 'y', 'z']
>>> for x in chain(a, b):
...     print(x)
...
1
2
3
4
x
y
z
>>>
```

使用 `chain()` 的一个常见场景是当你想对不同的集合中所有元素执行某些操作的时候。比如：

```
# Various working sets of items
active_items = set()
inactive_items = set()

# Iterate over all items
for item in chain(active_items, inactive_items):
    # Process item
```

这种解决方案要比像下面这样使用两个单独的循环更加优雅，

```
for item in active_items:
    # Process item
    ...

for item in inactive_items:
    # Process item
    ...
```

讨论

`itertools.chain()` 接受一个或多个可迭代对象最为输入参数。然后创建一个迭代器，依次连续的返回每个可迭代对象中的元素。这种方式要比先将序列合并再迭代要高效的多。比如：

```
# Inefficient
for x in a + b:
    ...

# Better
for x in chain(a, b):
    ...
```

第一种方案中，`a + b` 操作会创建一个全新的序列并要求a和b的类型一致。`chain()` 不会有这一步，所以如果输入序列非常大的时候会很省内存。并且当可迭代对象类型不一样的时候 `chain()` 同样可以很好的工作。

4.13 创建数据处理管道

问题

你想以数据管道(类似Unix管道)的方式迭代处理数据。比如，你有个大量的数据需要处理，但是不能将它们一次性放入内存中。

解决方案

生成器函数是一个实现管道机制的好办法。为了演示，假定你要处理一个非常大的日志文件目录：

```
foo/
  access-log-012007.gz
  access-log-022007.gz
  access-log-032007.gz
  ...
  access-log-012008
bar/
  access-log-092007.bz2
  ...
  access-log-022008
```

假设每个日志文件包含这样的数据：

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -
...
```

为了处理这些文件，你可以定义一个由多个执行特定任务独立任务的简单生成器函数组成的容器。就像这样：

```

import os
import fnmatch
import gzip
import bz2
import re

def gen_find(filepat, top):
    """
    Find all filenames in a directory tree that match a shell wildcard pattern
    """
    for path, dirlist, filelist in os.walk(top):
        for name in fnmatch.filter(filelist, filepat):
            yield os.path.join(path,name)

def gen_opener(filenamees):
    """
    Open a sequence of filenames one at a time producing a file object.
    The file is closed immediately when proceeding to the next iteration.
    """
    for filename in filenamees:
        if filename.endswith('.gz'):
            f = gzip.open(filename, 'rt')
        elif filename.endswith('.bz2'):
            f = bz2.open(filename, 'rt')
        else:
            f = open(filename, 'rt')
        yield f
        f.close()

def gen_concatenate(iterators):
    """
    Chain a sequence of iterators together into a single sequence.
    """
    for it in iterators:
        yield from it

def gen_grep(pattern, lines):
    """
    Look for a regex pattern in a sequence of lines
    """
    pat = re.compile(pattern)
    for line in lines:
        if pat.search(line):
            yield line

```

现在你可以很容易的将这些函数连起来创建一个处理管道。比如，为了查找包含单词python的所有日志行，你可以这样做：

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?!i)python', lines)
for line in pylines:
    print(line)
```

如果将来的时候你想扩展管道，你甚至可以在生成器表达式中包装数据。比如，下面这个版本计算出传输的字节数并计算其总和。

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?!i)python', lines)
bytecolumn = (line.rsplit(None,1)[1] for line in pylines)
bytes = (int(x) for x in bytecolumn if x != '-')
print('Total', sum(bytes))
```

讨论

以管道方式处理数据可以用来解决各类其他问题，包括解析，读取实时数据，定时轮询等。

为了理解上述代码，重点是要明白 `yield` 语句作为数据的生产者而 `for` 循环语句作为数据的消费者。当这些生成器被连在一起后，每个 `yield` 会将一个单独的数据元素传递给迭代处理管道的下一阶段。在例子最后部分，`sum()` 函数是最终的程序驱动者，每次从生成器管道中提取出一个元素。

这种方式一个非常好的特点是每个生成器函数很小并且都是独立的。这样的话就很容易编写和维护它们了。很多时候，这些函数如果比较通用的话可以在其他场景重复使用。并且最终将这些组件组合起来的代码看上去非常简单，也很容易理解。

使用这种方式的内存效率也不得不提。上述代码即便是在一个超大型文件目录中也能工作的很好。事实上，由于使用了迭代方式处理，代码运行过程中只需要很小很小的内存。

在调用 `gen_concatenate()` 函数的时候你可能会有些不太明白。这个函数的目的是将输入序列拼接成一个很长的行序列。`itertools.chain()` 函数同样有类似的功能，但是它需要将所有可迭代对象最为参数传入。在上面这个例子中，你可能会写类似这样的语句

`lines = itertools.chain(*files)`，使得 `gen_opener()` 生成器能被全部消费掉。但由于 `gen_opener()` 生成器每次生成一个打开过的文件，等到下一个迭代步骤时文件就关闭了，因此 `china()` 在这里不能这样使用。上面的方案可以避免这种情况。

`gen_concatenate()` 函数中出现过 `yield from` 语句，它将 `yield` 操作代理到父生成器上去。语句 `yield from it` 简单的返回生成器 `it` 所产生的所有值。关于这个我们在4.14小节会有更进一步的描述。

最后还有一点需要注意的是，管道方式并不是万能的。有时候你想立即处理所有数据。然而，即便是这种情况，使用生成器管道也可以将这类问题从逻辑上变为工作流的处理方式。

David Beazley 在他的 [Generator Tricks for Systems Programmers](#) 教程中对于这种技术有非常深入的讲解。可以参考这个教程获取更多的信息。

4.14 展开嵌套的序列

问题

你想将一个多层嵌套的序列展开成一个单层列表

解决方案

可以写一个包含 `yield from` 语句的递归生成器来轻松解决这个问题。比如：

```
from collections import Iterable

def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            yield from flatten(x)
        else:
            yield x

items = [1, 2, [3, 4, [5, 6], 7], 8]
# Produces 1 2 3 4 5 6 7 8
for x in flatten(items):
    print(x)
```

在上面代码中，`isinstance(x, Iterable)` 检查某个元素是否是可迭代的。如果是的话，`yield from` 就会返回所有子例程的值。最终返回结果就是一个没有嵌套的简单序列了。

额外的参数 `ignore_types` 和检测语句 `isinstance(x, ignore_types)` 用来将字符串和字节排除在可迭代对象外，防止将它们再展开成单个的字符。这样的话字符串数组就能最终返回我们所期望的结果了。比如：

```
>>> items = ['Dave', 'Paula', ['Thomas', 'Lewis']]
>>> for x in flatten(items):
...     print(x)
...
Dave
Paula
Thomas
Lewis
>>>
```

讨论

语句 `yield from` 在你想在生成器中调用其他生成器作为子例程的时候非常有用。如果你不使用它的话，那么就必须写额外的 `for` 循环了。比如：

```
def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            for i in flatten(x):
                yield i
        else:
            yield x
```

尽管只改了一点点，但是 `yield from` 语句看上去感觉更好，并且也使得代码更简洁清爽。

之前提到的对于字符串和字节的额外检查是为了防止将它们再展开成单个字符。如果还有其他你不想展开的类型，修改参数 `ignore_types` 即可。

最后要注意的一点是，`yield from` 在涉及到基于协程和生成器的并发编程中扮演着更加重要的角色。可以参考12.12小节查看另外一个例子。

4.15 顺序迭代合并后的排序迭代对象

问题

你有一系列排序序列，想将它们合并后得到一个排序序列并在上面迭代遍历。

解决方案

`heapq.merge()` 函数可以帮你解决这个问题。比如：

```
>>> import heapq
>>> a = [1, 4, 7, 10]
>>> b = [2, 5, 6, 11]
>>> for c in heapq.merge(a, b):
...     print(c)
...
1
2
4
5
6
7
10
11
```

讨论

`heapq.merge` 可迭代特性意味着它不会立马读取所有序列。这就意味着你可以在非常长的序列中使用它，而不会有太大的开销。比如，下面是一个例子来演示如何合并两个排序文件：

```
with open('sorted_file_1', 'rt') as file1, \
     open('sorted_file_2', 'rt') as file2, \
     open('merged_file', 'wt') as outf:

    for line in heapq.merge(file1, file2):
        outf.write(line)
```

有一点要强调的是 `heapq.merge()` 需要所有输入序列必须是排过序的。特别的，它并不会预先读取所有数据到堆栈中或者预先排序，也不会对输入做任何的排序检测。它仅仅是检查所有序列的开始部分并返回最小的那个，这个过程一直会持续直到所有输入序列中的元素都被遍历完。

4.16 迭代器代替 **while** 无限循环

问题

你在代码中使用 `while` 循环来迭代处理数据，因为它需要调用某个函数或者和一般迭代模式不同的测试条件。能不能用迭代器来重写这个循环呢？

解决方案

一个常见的IO操作程序可能会想下面这样：

```
CHUNKSIZE = 8192

def reader(s):
    while True:
        data = s.recv(CHUNKSIZE)
        if data == b'':
            break
        process_data(data)
```

这种代码通常可以使用 `iter()` 来代替，如下所示：

```
def reader2(s):
    for chunk in iter(lambda: s.recv(CHUNKSIZE), b''):
        pass
    # process_data(data)
```

如果你怀疑它到底能不能正常工作，可以试验下一个简单的例子。比如：

```
>>> import sys
>>> f = open('/etc/passwd')
>>> for chunk in iter(lambda: f.read(10), ''):
...     n = sys.stdout.write(chunk)
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/uucico
...
>>>
```

讨论

`iter` 函数一个鲜为人知的特性是它接受一个可选的 `callable` 对象和一个标记(结尾)值作为输入参数。当以这种方式使用的时候，它会创建一个迭代器，这个迭代器会不断调用 `callable` 对象直到返回值和标记值相等为止。

这种特殊的方法对于一些特定的会被重复调用的函数很有效果，比如涉及到I/O调用的函数。举例来讲，如果你想从套接字或文件中以数据块的方式读取数据，通常你得要不断重复的执行 `read()` 或 `recv()`，并在后面紧跟一个文件结尾测试来决定是否终止。这节中的方案使用一个简单的 `iter()` 调用就可以将两者结合起来了。其中 `lambda` 函数参数是为了创建一个无参的 `callable` 对象，并为 `recv` 或 `read()` 方法提供了 `size` 参数。

第五章：文件与IO

所有程序都要处理输入和输出。这一章将涵盖处理不同类型的文件，包括文本和二进制文件，文件编码和其他相关的内容。对文件名和目录的操作也会涉及到。

Contents:

5.1 读写文本数据

问题

你需要读写各种不同编码的文本数据，比如ASCII，UTF-8或UTF-16编码等。

解决方案

使用带有 `rt` 模式的 `open()` 函数读取文本文件。如下所示：

```
# Read the entire file as a single string
with open('somefile.txt', 'rt') as f:
    data = f.read()

# Iterate over the lines of the file
with open('somefile.txt', 'rt') as f:
    for line in f:
        # process line
        ...
```

类似的，为了写入一个文本文件，使用带有 `wt` 模式的 `open()` 函数，如果之前文件内容存在则清除并覆盖掉。如下所示：

```
# Write chunks of text data
with open('somefile.txt', 'wt') as f:
    f.write(text1)
    f.write(text2)
    ...

# Redirected print statement
with open('somefile.txt', 'wt') as f:
    print(line1, file=f)
    print(line2, file=f)
    ...
```

如果是在已存在文件中添加内容，使用模式为 `at` 的 `open()` 函数。

文件的读写操作默认使用系统编码，可以通过调用 `sys.getdefaultencoding()` 来得到。在大多数机器上面都是utf-8编码。如果你已经知道你要读写的文本是其他编码方式，那么可以通过传递一个可选的 `encoding` 参数给`open()`函数。如下所示：

```
with open('somefile.txt', 'rt', encoding='latin-1') as f:
    ...
```

Python支持非常多的文本编码。几个常见的编码是ascii, latin-1, utf-8和utf-16。在web应用程序中通常都使用的是UTF-8。ascii对应从U+0000到U+007F范围内的7位字符。latin-1是字节0-255到U+0000至U+00FF范围内Unicode字符的直接映射。当读取一个未知编码的文本时使用latin-1编码永远不会产生解码错误。使用latin-1编码读取一个文件的时候也许不能产生完全正确的文本解码数据，但是它也能从中提取出足够多的有用数据。同时，如果你之后将数据回写回去，原先的数据还是会保留的。

讨论

读写文本文件一般来讲是比较简单的。但是也几点是需要注意的。首先，在例子程序中的with语句给被使用到的文件创建了一个上下文环境，但 `with` 控制块结束时，文件会自动关闭。你也可以不使用 `with` 语句，但是这时候你就必须记得手动关闭文件：

```
f = open('somefile.txt', 'rt')
data = f.read()
f.close()
```

另外一个是关于换行符的识别问题，在Unix和Windows中是不一样的(分别是n和rn)。默认情况下，Python会以统一模式处理换行符。这种模式下，在读取文本的时候，Python可以识别所有的普通换行符并将其转换为单个 `\n` 字符。类似的，在输出时会将换行符 `\n` 转换为系统默认的换行符。如果你不希望这种默认的处理方式，可以给 `open()` 函数传入参数 `newline=''`，就像下面这样：

```
# Read with disabled newline translation
with open('somefile.txt', 'rt', newline='') as f:
    ...
```

为了说明两者之间的差异，下面我在Unix机器上面读取一个Windows上面的文本文件，里面的内容是 `hello world!\r\n`：

```
>>> # Newline translation enabled (the default)
>>> f = open('hello.txt', 'rt')
>>> f.read()
'hello world!\n'

>>> # Newline translation disabled
>>> g = open('hello.txt', 'rt', newline='')
>>> g.read()
'hello world!\r\n'
>>>
```

最后一个问题就是文本文件中可能出现的编码错误。但你读取或者写入一个文本文件时，你可能会遇到一个编码或者解码错误。比如：

```
>>> f = open('sample.txt', 'rt', encoding='ascii')
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/encodings/ascii.py", line 26, in decode
    return codecs.ascii_decode(input, self.errors)[0]
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position
12: ordinal not in range(128)
>>>
```

如果出现这个错误，通常表示你读取文本时指定的编码不正确。你最好仔细阅读说明并确认你的文件编码是正确的(比如使用UTF-8而不是Latin-1编码或其他)。如果编码错误还是存在的话，你可以给 `open()` 函数传递一个可选的 `errors` 参数来处理这些错误。下面是一些处理常见错误的方法：

```
>>> # Replace bad chars with Unicode U+fffd replacement char
>>> f = open('sample.txt', 'rt', encoding='ascii', errors='replace')
>>> f.read()
'Spicy Jalapeño!'
>>> # Ignore bad chars entirely
>>> g = open('sample.txt', 'rt', encoding='ascii', errors='ignore')
>>> g.read()
'Spicy Jalapeo!'
>>>
```

如果你经常使用 `errors` 参数来处理编码错误，可能会让你的生活变得很糟糕。对于文本处理的首要原则是确保你总是使用的是正确编码。当模棱两可的时候，就使用默认的设置(通常都是UTF-8)。

5.2 打印输出至文件中

问题

你想将 `print()` 函数的输出重定向到一个文件中去。

解决方案

在 `print()` 函数中指定 `file` 关键字参数，像下面这样：

```
with open('d:/work/test.txt', 'wt') as f:
    print('Hello World!', file=f)
```

讨论

关于输出重定向到文件中就这些了。但是有一点要注意的就是文件必须是以文本模式打开。如果文件是二进制模式的话，打印就会出错。

5.3 使用其他分隔符或行终止符打印

问题

你想使用 `print()` 函数输出数据，但是想改变默认的分隔符或者行尾符。

解决方案

可以使用在 `print()` 函数中使用 `sep` 和 `end` 关键字参数，以你想要的方式输出。比如：

```
>>> print('ACME', 50, 91.5)
ACME 50 91.5
>>> print('ACME', 50, 91.5, sep=',')
ACME,50,91.5
>>> print('ACME', 50, 91.5, sep=',', end='!!\n')
ACME,50,91.5!!
>>>
```

使用 `end` 参数也可以在输出中禁止换行。比如：

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> for i in range(5):
...     print(i, end=' ')
...
0 1 2 3 4 >>>
```

讨论

当你想使用非空格分隔符来输出数据的时候，给 `print()` 函数传递一个 `sep` 参数是最简单的方案。有时候你会看到一些程序员会使用 `str.join()` 来完成同样的事情。比如：

```
>>> print(','.join('ACME', '50', '91.5'))
ACME,50,91.5
>>>
```

`str.join()` 的问题在于它仅仅适用于字符串。这意味着你通常需要执行另外一些转换才能让它正常工作。比如：

```
>>> row = ('ACME', 50, 91.5)
>>> print(', '.join(row))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected str instance, int found
>>> print(', '.join(str(x) for x in row))
ACME,50,91.5
>>>
```

你当然可以不用那么麻烦，仅仅只需要像下面这样写：

```
>>> print(*row, sep=',')
ACME,50,91.5
>>>
```

5.4 读写字节数据

问题

你想读写二进制文件，比如图片，声音文件等等。

解决方案

使用模式为 `rb` 或 `wb` 的 `open()` 函数来读取或写入二进制数据。比如：

```
# Read the entire file as a single byte string
with open('somefile.bin', 'rb') as f:
    data = f.read()

# Write binary data to a file
with open('somefile.bin', 'wb') as f:
    f.write(b'Hello World')
```

在读取二进制数据时，需要指明的是所有返回的数据都是字节字符串格式的，而不是文本字符串。类似的，在写入的时候，必须保证参数是以字节形式对外暴露数据的对象(比如字节字符串，字节数组对象等)。

讨论

在读取二进制数据的时候，字节字符串和文本字符串的语义差异可能会导致一个潜在的陷阱。特别需要注意的是，索引和迭代动作返回的是字节的值而不是字节字符串。比如：

```
>>> # Text string
>>> t = 'Hello World'
>>> t[0]
'H'
>>> for c in t:
...     print(c)
...
H
e
l
l
o
...
>>> # Byte string
>>> b = b'Hello World'
>>> b[0]
72
>>> for c in b:
...     print(c)
...
72
101
108
108
111
...
>>>
```

如果你想从二进制模式的文件中读取或写入文本数据，必须确保要进行解码和编码操作。比如：

```
with open('somefile.bin', 'rb') as f:
    data = f.read(16)
    text = data.decode('utf-8')

with open('somefile.bin', 'wb') as f:
    text = 'Hello World'
    f.write(text.encode('utf-8'))
```

二进制I/O还有一个鲜为人知的特性就是数组和C结构体类型能直接被写入，而不需要中间转换为自己对象。比如：

```
import array
nums = array.array('i', [1, 2, 3, 4])
with open('data.bin', 'wb') as f:
    f.write(nums)
```

这个适用于任何实现了被称之为“缓冲接口”的对象，这种对象会直接暴露其底层的内存缓冲区给能处理它的操作。二进制数据的写入就是这类操作之一。

很多对象还允许通过使用文件对象的 `readinto()` 方法直接读取二进制数据到其底层的内存中去。比如：

```
>>> import array
>>> a = array.array('i', [0, 0, 0, 0, 0, 0, 0, 0])
>>> with open('data.bin', 'rb') as f:
...     f.readinto(a)
...
16
>>> a
array('i', [1, 2, 3, 4, 0, 0, 0, 0])
>>>
```

但是使用这种技术的时候需要格外小心，因为它通常具有平台相关性，并且可能会依赖字长和字节顺序(高位优先和低位优先)。可以查看5.9小节中另外一个读取二进制数据到可修改缓冲区的例子。

5.5 文件不存在才能写入

问题

你想像一个文件中写入数据，但是前提必须是这个文件在文件系统上不存在。也就是不允许覆盖已存在的文件内容。

解决方案

可以在 `open()` 函数中使用 `x` 模式来代替 `w` 模式的方法来解决这个问题。比如：

```
>>> with open('somefile', 'wt') as f:
...     f.write('Hello\n')
...
>>> with open('somefile', 'xt') as f:
...     f.write('Hello\n')
...
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'somefile'
>>>
```

如果文件是二进制的，使用 `xb` 来代替 `xt`

讨论

这一小节演示了在写文件时通常会遇到的一个问题的完美解决方案(不小心覆盖一个已存在的文件)。一个替代方案是先测试这个文件是否存在，像下面这样：

```
>>> import os
>>> if not os.path.exists('somefile'):
...     with open('somefile', 'wt') as f:
...         f.write('Hello\n')
... else:
...     print('File already exists!')
...
File already exists!
>>>
```

显而易见，使用x文件模式更加简单。要注意的是x模式是一个Python3对 `open()` 函数特有的扩展。在Python的旧版本或者是Python实现的底层C函数库中都是没有这个模式的。

5.6 字符串的I/O操作

问题

你想使用操作类文件对象的程序来操作文本或二进制字符串。

解决方案

使用 `io.StringIO()` 和 `io.BytesIO()` 类来创建类文件对象操作字符串数据。比如：

```
>>> s = io.StringIO()
>>> s.write('Hello World\n')
12
>>> print('This is a test', file=s)
15
>>> # Get all of the data written so far
>>> s.getvalue()
'Hello World\nThis is a test\n'
>>>

>>> # Wrap a file interface around an existing string
>>> s = io.StringIO('Hello\nWorld\n')
>>> s.read(4)
'Hell'
>>> s.read()
'o\nWorld\n'
>>>
```

`io.StringIO` 只能用于文本。如果你要操作二进制数据，要使用 `io.BytesIO` 类来代替。比如：

```
>>> s = io.BytesIO()
>>> s.write(b'binary data')
>>> s.getvalue()
b'binary data'
>>>
```

讨论

当你想模拟一个普通的文件的时候 `StringIO` 和 `BytesIO` 类是很有用的。比如，在单元测试中，你可以使用 `StringIO` 来创建一个包含测试数据的类文件对象，这个对象可以被传给某个参数为普通文件对象的函数。

需要注意的是，`StringIO` 和 `BytesIO` 实例并没有正确的整数类型的文件描述符。因此，它们不能在那些需要使用真实的系统级文件如文件，管道或者是套接字的程序中使用。

5.7 读写压缩文件

问题

你想读写一个gzip或bz2格式的压缩文件。

解决方案

`gzip` 和 `bz2` 模块可以很容易的处理这些文件。两个模块都为 `open()` 函数提供了另外的实现来解决这个问题。比如，为了以文本形式读取压缩文件，可以这样做：

```
# gzip compression
import gzip
with gzip.open('somefile.gz', 'rt') as f:
    text = f.read()

# bz2 compression
import bz2
with bz2.open('somefile.bz2', 'rt') as f:
    text = f.read()
```

类似的，为了写入压缩数据，可以这样做：

```
# gzip compression
import gzip
with gzip.open('somefile.gz', 'wt') as f:
    f.write(text)

# bz2 compression
import bz2
with bz2.open('somefile.bz2', 'wt') as f:
    f.write(text)
```

如上，所有的I/O操作都使用文本模式并执行Unicode的编码/解码。类似的，如果你想操作二进制数据，使用 `rb` 或者 `wb` 文件模式即可。

讨论

大部分情况下读写压缩数据都是很简单的。但是要注意的是选择一个正确的文件模式是非常重要的。如果你不指定模式，那么默认的就是二进制模式，如果这时候程序想要接受的是文本数据，那么就会出错。`gzip.open()` 和 `bz2.open()` 接受跟内置的 `open()` 函数一样的参数，包括 `encoding`，`errors`，`newline` 等等。

当写入压缩数据时，可以使用 `compresslevel` 这个可选的关键字参数来指定一个压缩级别。比如：

```
with gzip.open('somefile.gz', 'wt', compresslevel=5) as f:
    f.write(text)
```

默认的等级是9，也是最高的压缩等级。等级越低性能越好，但是数据压缩程度也越低。

最后一点，`gzip.open()` 和 `bz2.open()` 还有一个很少被知道的特性，它们可以作用在一个已存在并以二进制模式打开的文件上。比如，下面代码是可行的：

```
import gzip
f = open('somefile.gz', 'rb')
with gzip.open(f, 'rt') as g:
    text = g.read()
```

这样就允许 `gzip` 和 `bz2` 模块可以工作在许多类文件对象上，比如套接字，管道和内存中文件等。

5.8 固定大小记录的文件迭代

问题

你想在一个固定长度记录或者数据块的集合上迭代，而不是在一个文件中一行一行的迭代。

解决方案

通过下面这个小技巧使用 `iter` 和 `functools.partial()` 函数：

```
from functools import partial

RECORD_SIZE = 32

with open('somefile.data', 'rb') as f:
    records = iter(partial(f.read, RECORD_SIZE), b'')
    for r in records:
        ...
```

这个例子中的 `records` 对象是一个可迭代对象，它会不断的产生固定大小的数据块，直到文件末尾。要注意的是如果总记录大小不是块大小的整数倍的话，最后一个返回元素的字节数会比期望值少。

讨论

`iter()` 函数有一个鲜为人知的特性就是，如果你给它传递一个可调用对象和一个标记值，它会创建一个迭代器。这个迭代器会一直调用传入的可调用对象直到它返回标记值为止，这时候迭代终止。

在例子中，`functools.partial` 用来创建一个每次被调用时从文件中读取固定数目字节的可调用对象。标记值 `b''` 就是当到达文件结尾时的返回值。

最后再提一点，上面的例子中的文件是以二进制模式打开的。如果是读取固定大小的记录，这通常是最普遍的情况。而对于文本文件，一行一行的读取(默认的迭代行为)更普遍点。

5.9 读取二进制数据到可变缓冲区中

问题

你想直接读取二进制数据到一个可变缓冲区中，而不需要做任何中间复制操作。或者你想原地修改数据并将它写回到一个文件中去。

解决方案

为了读取数据到一个可变数组中，使用文件对象的 `readinto()` 方法。比如：

```
import os.path

def read_into_buffer(filename):
    buf = bytearray(os.path.getsize(filename))
    with open(filename, 'rb') as f:
        f.readinto(buf)
    return buf
```

下面是一个演示这个函数使用方法的例子：

```

>>> # Write a sample file
>>> with open('sample.bin', 'wb') as f:
...     f.write(b'Hello World')
...
>>> buf = read_into_buffer('sample.bin')
>>> buf
bytearray(b'Hello World')
>>> buf[0:5] = b'Hallo'
>>> buf
bytearray(b'Hallo World')
>>> with open('newsample.bin', 'wb') as f:
...     f.write(buf)
...
11
>>>

```

讨论

文件对象的 `readinto()` 方法能被用来为预先分配内存的数组填充数据，甚至包括由 `array` 模块或 `numpy` 库创建的数组。和普通 `read()` 方法不同的是，`readinto()` 填充已存在的缓冲区而不是为新对象重新分配内存再返回它们。因此，你可以使用它来避免大量的内存分配操作。比如，如果你读取一个由相同大小的记录组成的二进制文件时，你可以像下面这样写：

```

record_size = 32 # Size of each record (adjust value)

buf = bytearray(record_size)
with open('somefile', 'rb') as f:
    while True:
        n = f.readinto(buf)
        if n < record_size:
            break
        # Use the contents of buf
        ...

```

另外有一个有趣特性就是 `memoryview`，它可以通过零复制的方式对已存在的缓冲区执行切片操作，甚至还能修改它的内容。比如：

```
>>> buf
bytearray(b'Hello World')
>>> m1 = memoryview(buf)
>>> m2 = m1[-5:]
>>> m2
<memory at 0x100681390>
>>> m2[:] = b'WORLD'
>>> buf
bytearray(b'Hello WORLD')
>>>
```

使用 `f.readinto()` 时需要注意的是，你必须检查它的返回值，也就是实际读取的字节数。

如果字节数小于缓冲区大小，表明数据被截断或者被破坏了(比如你期望每次读取指定数量的字节)。

最后，留心观察其他函数库和模块中和 `into` 相关的函数(比如 `recv_into()`，`pack_into()` 等)。Python的很多其他部分已经能支持直接的I/O或数据访问操作，这些操作可被用来填充或修改数组和缓冲区内容。

关于解析二进制结构和 `memoryviews` 使用方法的更高级例子，请参考6.12小节。

5.10 内存映射的二进制文件

问题

你想内存映射一个二进制文件到一个可变字节数组中，目的可能是为了随机访问它的内容或者是原地做些修改。

解决方案

使用 `mmap` 模块来内存映射文件。下面是一个工具函数，向你演示了如何打开一个文件并以一种便捷方式内存映射这个文件。

```
import os
import mmap

def memory_map(filename, access=mmap.ACCESS_WRITE):
    size = os.path.getsize(filename)
    fd = os.open(filename, os.O_RDWR)
    return mmap.mmap(fd, size, access=access)
```

为了使用这个函数，你需要有一个已创建并且内容不为空的文件。下面是一个例子，教你怎样初始创建一个文件并将其内容扩充到指定大小：

```
>>> size = 1000000
>>> with open('data', 'wb') as f:
...     f.seek(size-1)
...     f.write(b'\x00')
...
>>>
```

下面是一个利用 `memory_map()` 函数类内存映射文件内容的例子：

```
>>> m = memory_map('data')
>>> len(m)
1000000
>>> m[0:10]
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> m[0]
0
>>> # Reassign a slice
>>> m[0:11] = b'Hello World'
>>> m.close()

>>> # Verify that changes were made
>>> with open('data', 'rb') as f:
...     print(f.read(11))
...
b'Hello World'
>>>
```

`mmap()` 返回的 `mmap` 对象同样也可以作为一个上下文管理器来使用，这时候底层的文件会被自动关闭。比如：

```
>>> with memory_map('data') as m:
...     print(len(m))
...     print(m[0:10])
...
1000000
b'Hello World'
>>> m.closed
True
>>>
```

默认情况下，`memeory_map()` 函数打开的文件同时支持读和写操作。任何的修改内容都会复制回原来的文件中。如果需要只读的访问模式，可以给参数 `access` 赋值为 `mmap.ACCESS_READ`。比如：

```
m = memory_map(filename, mmap.ACCESS_READ)
```

如果你想在本地修改数据，但是又不想将修改写回到原始文件中，可以使用

`mmap.ACCESS_COPY`：

```
m = memory_map(filename, mmap.ACCESS_COPY)
```

讨论

为了随机访问文件的内容，使用 `mmap` 将文件映射到内存中是一个高效和优雅的方法。例如，你无需打开一个文件并执行大量的 `seek()`，`read()`，`write()` 调用，只需要简单的映射文件并使用切片操作访问数据即可。

一般来讲，`mmap()` 所暴露的内存看上去就是一个二进制数组对象。但是，你可以使用一个内存视图来解析其中的数据。比如：

```
>>> m = memory_map('data')
>>> # Memoryview of unsigned integers
>>> v = memoryview(m).cast('I')
>>> v[0] = 7
>>> m[0:4]
b'\x07\x00\x00\x00'
>>> m[0:4] = b'\x07\x01\x00\x00'
>>> v[0]
263
>>>
```

需要强调的一点是，内存映射一个文件并不会导致整个文件被读取到内存中。也就是说，文件并没有被复制到内存缓存或数组中。相反，操作系统仅仅为文件内容保留了一段虚拟内存。当你访问文件的不同区域时，这些区域的内容才根据需要被读取并映射到内存区域中。而那些从没被访问到的部分还是留在磁盘上。所有这些过程是透明的，在幕后完成！

如果多个Python解释器内存映射同一个文件，得到的 `mmap` 对象能够被用来在解释器直接交换数据。也就是说，所有解释器都能同时读写数据，并且其中一个解释器所做的修改会自动呈现在其他解释器中。很明显，这里需要考虑同步的问题。但是这种方法有时候可以用来在管道或套接字间传递数据。

这一小节中函数尽量写得很通用，同时适用于Unix和Windows平台。要注意的是使用 `mmap()` 函数时会在底层有一些平台的差异性。另外，还有一些选项可以用来创建匿名的内存映射区域。如果你对这个感兴趣，确保你仔细研读了Python文档中 [这方面的内容](#)。

5.11 文件路径名的操作

问题

你需要使用路径名来获取文件名，目录名，绝对路径等等。

解决方案

使用 `os.path` 模块中的函数来操作路径名。下面是一个交互式例子来演示一些关键的特性：

```
>>> import os
>>> path = '/Users/beazley/Data/data.csv'

>>> # Get the last component of the path
>>> os.path.basename(path)
'data.csv'

>>> # Get the directory name
>>> os.path.dirname(path)
'/Users/beazley/Data'

>>> # Join path components together
>>> os.path.join('tmp', 'data', os.path.basename(path))
'tmp/data/data.csv'

>>> # Expand the user's home directory
>>> path = '~/Data/data.csv'
>>> os.path.expanduser(path)
'/Users/beazley/Data/data.csv'

>>> # Split the file extension
>>> os.path.splitext(path)
('~'/Data/data', '.csv')
>>>
```

讨论

对于任何的文件名的操作，你都应该使用 `os.path` 模块，而不是使用标准字符串操作来构造自己的代码。特别是为了可移植性考虑的时候更应如此，因为 `os.path` 模块知道 Unix和Windows系统之间的差异并且能够可靠地处理类似 `Data/data.csv` 和 `Data\data.csv` 这样的文件名。其次，你真的不应该浪费时间去重复造轮子。通常最好是直接使用已经为你准备好的功能。

要注意的是 `os.path` 还有更多的功能在这里并没有列举出来。可以查阅官方文档来获取更多与文件测试，符号链接等相关的函数说明。

5.12 测试文件是否存在

问题

你想测试一个文件或目录是否存在。

解决方案

使用 `os.path` 模块来测试一个文件或目录是否存在。比如：

```
>>> import os
>>> os.path.exists('/etc/passwd')
True
>>> os.path.exists('/tmp/spam')
False
>>>
```

你还能进一步测试这个文件时什么类型的。在下面这些测试中，如果测试的文件不存在的时候，结果都会返回False：

```
>>> # Is a regular file
>>> os.path.isfile('/etc/passwd')
True

>>> # Is a directory
>>> os.path.isdir('/etc/passwd')
False

>>> # Is a symbolic link
>>> os.path.islink('/usr/local/bin/python3')
True

>>> # Get the file linked to
>>> os.path.realpath('/usr/local/bin/python3')
'/usr/local/bin/python3.3'
>>>
```

如果你还想获取元数据(比如文件大小或者是修改日期), 也可以使用 `os.path` 模块来解决:

```
>>> os.path.getsize('/etc/passwd')
3669
>>> os.path.getmtime('/etc/passwd')
1272478234.0
>>> import time
>>> time.ctime(os.path.getmtime('/etc/passwd'))
'Wed Apr 28 13:10:34 2010'
>>>
```

讨论

使用 `os.path` 来进行文件测试是很简单的。在写这些脚本时, 可能唯一需要注意的就是你需要考虑文件权限的问题, 特别是在获取元数据时候。比如:

```
>>> os.path.getsize('/Users/guido/Desktop/foo.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/genericpath.py", line 49, in getsize
    return os.stat(filename).st_size
PermissionError: [Errno 13] Permission denied: '/Users/guido/Desktop/foo.txt'
>>>
```

5.13 获取文件夹中的文件列表

问题

你想获取文件系统中某个目录下的所有文件列表。

解决方案

使用 `os.listdir()` 函数来获取某个目录中的文件列表：

```
import os
names = os.listdir('somedir')
```

结果会返回目录中所有文件列表，包括所有文件，子目录，符号链接等等。如果你需要通过某种方式过滤数据，可以考虑结合 `os.path` 库中的一些函数来使用列表推导。比如：

```
import os.path

# Get all regular files
names = [name for name in os.listdir('somedir')
         if os.path.isfile(os.path.join('somedir', name))]

# Get all dirs
dirnames = [name for name in os.listdir('somedir')
            if os.path.isdir(os.path.join('somedir', name))]
```

字符串的 `startswith()` 和 `endswith()` 方法对于过滤一个目录的内容也是很有用的。比如：

```
pyfiles = [name for name in os.listdir('somedir')
           if name.endswith('.py')]
```

对于文件名的匹配，你可能会考虑使用 `glob` 或 `fnmatch` 模块。比如：

```
import glob
pyfiles = glob.glob('somedir/*.py')

from fnmatch import fnmatch
pyfiles = [name for name in os.listdir('somedir')
           if fnmatch(name, '*.py')]
```

讨论

获取目录中的列表是很容易的，但是其返回结果只是目录中实体名列表而已。如果你还想获取其他的元信息，比如文件大小，修改时间等等，你或许还需要使用到 `os.path` 模块中的函数或者 `os.stat()` 函数来收集数据。比如：

```
# Example of getting a directory listing

import os
import os.path
import glob

pyfiles = glob.glob('*.py')

# Get file sizes and modification dates
name_sz_date = [(name, os.path.getsize(name), os.path.getmtime(name))
                 for name in pyfiles]
for name, size, mtime in name_sz_date:
    print(name, size, mtime)

# Alternative: Get file metadata
file_metadata = [(name, os.stat(name)) for name in pyfiles]
for name, meta in file_metadata:
    print(name, meta.st_size, meta.st_mtime)
```

最后还有一点要注意的就是，有时候在处理文件名编码问题时候可能会出现一些问题。通常来讲，函数 `os.listdir()` 返回的实体列表会根据系统默认的文件名编码来解码。但是有时候也会碰到一些不能正常解码的文件名。关于文件名的处理问题，在5.14和5.15小节有更详细的讲解。

5.14 忽略文件名编码

问题

你想使用原始文件名执行文件的I/O操作，也就是说文件名并没有经过系统默认编码去解码或编码过。

解决方案

默认情况下，所有的文件名都会根据 `sys.getfilesystemencoding()` 返回的文本编码来编码或解码。比如：

```
>>> sys.getfilesystemencoding()
'utf-8'
>>>
```

如果因为某种原因你想忽略这种编码，可以使用一个原始字节字符串来指定一个文件名即可。比如：

```
>>> # Write a file using a unicode filename
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('Spicy!')
...
6
>>> # Directory listing (decoded)
>>> import os
>>> os.listdir('.')
['jalape\u00f1o.txt']

>>> # Directory listing (raw)
>>> os.listdir(b'.') # Note: byte string
[b'jalapen\xcc\x83o.txt']

>>> # Open file with raw filename
>>> with open(b'jalapen\xcc\x83o.txt') as f:
...     print(f.read())
...
Spicy!
>>>
```

正如你所见，在最后两个操作中，当你给文件相关函数如 `open()` 和 `os.listdir()` 传递字节字符串时，文件名的处理方式会稍有不同。

讨论

通常来讲，你不需要担心文件名的编码和解码，普通的文件名操作应该就没问题了。但是，有些操作系统允许用户通过偶然或恶意方式去创建名字不符合默认编码的文件。这些文件名可能会神秘地中断那些需要处理大量文件的Python程序。

读取目录并通过原始未解码方式处理文件名可以有效的避免这样的问题，尽管这样会带来一定的编程难度。

关于打印不可解码的文件名，请参考5.15小节。

5.15 打印不合法的文件名

问题

你的程序获取了一个目录中的文件名列表，但是当它试着去打印文件名的时候程序崩溃，出现了 `UnicodeEncodeError` 异常和一条奇怪的消息—— `surrogates not allowed`。

解决方案

当打印未知的文件名时，使用下面的方法可以避免这样的错误：

```
def bad_filename(filename):
    return repr(filename)[1:-1]

try:
    print(filename)
except UnicodeEncodeError:
    print(bad_filename(filename))
```

讨论

这一小节讨论的是在编写必须处理文件系统的程序时一个不太常见但又很棘手的问题。默认情况下，Python假定所有文件名都已经根据 `sys.getfilesystemencoding()` 的值编码过了。但是，有一些文件系统并没有强制要求这样做，因此允许创建文件名没有正确编码的文件。这种情况不太常见，但是总会有些用户冒险这样做或者是无意之中这样做了(可能是在一个有缺陷的代码中给 `open()` 函数传递了一个不合规规范的文件名)。

当执行类似 `os.listdir()` 这样的函数时，这些不合规规范的文件名就会让Python陷入困境。一方面，它不能仅仅只是丢弃这些不合格的名字。而另一方面，它又不能将这些文件名转换为正确的文本字符串。Python对这个问题的解决方案是从文件名中获取未解码的字节值比如 `\xhh` 并将它映射成Unicode字符 `\udchh` 表示的所谓的“代理编码”。下面一个例子演示了当一个不合格目录列表中含有一个文件名为**bäd.txt**(使用Latin-1而不是UTF-8编码)时的样子：

```
>>> import os
>>> files = os.listdir('.')
>>> files
['spam.py', 'b\udce4d.txt', 'foo.txt']
>>>
```

如果你有代码需要操作文件名或者将文件名传递给 `open()` 这样的函数，一切都能正常工作。只有当你想要输出文件名时才会碰到些麻烦(比如打印输出到屏幕或日志文件等)。特别的，当你想打印上面的文件名列表时，你的程序就会崩溃：

```
>>> for name in files:
...     print(name)
...
spam.py
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udce4' in
position 1: surrogates not allowed
>>>
```

程序崩溃的原因就是字符 `\udce4` 是一个非法的Unicode字符。它其实是一个被称为代理字符对的双字符组合的后半部分。由于缺少了前半部分，因此它是个非法的Unicode。所以，唯一能成功输出的方法就是当遇到不合法文件名时采取相应的补救措施。比如可以将上述代码修改如下：

```
>>> for name in files:
...     try:
...         print(name)
...     except UnicodeEncodeError:
...         print(bad_filename(name))
...
spam.py
b\udce4d.txt
foo.txt
>>>
```

在 `bad_filename()` 函数中怎样处置取决于你自己。另外一个选择就是通过某种方式重新编码，示例如下：

```
def bad_filename(filename):
    temp = filename.encode(sys.getfilesystemencoding(), errors='surrogateescape')
    return temp.decode('latin-1')
```

译者注:

surrogateescape:

这种是Python在绝大部分面向OS的API中所使用的错误处理器，它能以一种优雅的方式处理由操作系统提供的数据的编码问题。在解码出错时会将出错字节存储到一个很少被使用到的Unicode编码范围内。在编码时将那些隐藏值又还原回原先解码失败的字节序列。它不仅对于OS API非常有用，也能很容易的处理其他情况下的编码错误。

使用这个版本产生的输出如下:

```
>>> for name in files:
...     try:
...         print(name)
...     except UnicodeEncodeError:
...         print(bad_filename(name))
...
spam.py
bäd.txt
foo.txt
>>>
```

这一小节主题可能会被大部分读者所忽略。但是如果你在编写依赖文件名和文件系统的关键任务程序时，就必须得考虑到这个。否则你可能会在某个周末被叫到办公室去调试一些令人费解的错误。

5.16 增加或改变已打开文件的编码

问题

你想在不关闭一个已打开的文件前提下增加或改变它的Unicode编码。

解决方案

如果你想给一个以二进制模式打开的文件添加Unicode编码/解码方式，可以使用

`io.TextIOWrapper()` 对象包装它。比如:

```
import urllib.request
import io

u = urllib.request.urlopen('http://www.python.org')
f = io.TextIOWrapper(u, encoding='utf-8')
text = f.read()
```

如果你想修改一个已经打开的文本模式的文件的编码方式，可以先使用 `detach()` 方法移除掉已存在的文本编码层，并使用新的编码方式代替。下面是一个在 `sys.stdout` 上修改编码方式的例子：

```
>>> import sys
>>> sys.stdout.encoding
'UTF-8'
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='latin-1')
>>> sys.stdout.encoding
'latin-1'
>>>
```

这样做可能会中断你的终端，这里仅仅是为了演示而已。

讨论

I/O系统由一系列的层次构建而成。你可以试着运行下面这个操作一个文本文件的例子来查看这种层次：

```
>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f.buffer
<_io.BufferedWriter name='sample.txt'>
>>> f.buffer.raw
<_io.FileIO name='sample.txt' mode='wb'>
>>>
```

在这个例子中，`io.TextIOWrapper` 是一个编码和解码Unicode的文本处理层，`io.BufferedWriter` 是一个处理二进制数据的带缓冲的I/O层，`io.FileIO` 是一个表示操作系统底层文件描述符的原始文件。增加或改变文本编码会涉及增加或改变最上面的 `io.TextIOWrapper` 层。

一般来讲，像上面例子这样通过访问属性值来直接操作不同的层是很不安全的。例如，如果你试着使用下面这样的技术改变编码看看会发生什么：

```
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f = io.TextIOWrapper(f.buffer, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>> f.write('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

结果出错了，因为f的原始值已经被破坏了并关闭了底层的文件。

`detach()` 方法会断开文件的最顶层并返回第二层，之后最顶层就没什么用了。例如：

```
>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> b = f.detach()
>>> b
<_io.BufferedWriter name='sample.txt'>
>>> f.write('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: underlying buffer has been detached
>>>
```

一旦断开最顶层后，你就可以给返回结果添加一个新的最顶层。比如：

```
>>> f = io.TextIOWrapper(b, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>>
```

尽管已经向你演示了改变编码的方法，但是你还可以利用这种技术来改变文件行处理、错误机制以及文件处理的其他方面。例如：


```
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='ascii',
...                               errors='xmlcharrefreplace')
>>> print('Jalape\u00f1o')
Jalape&#241;o
>>>
```

注意下最后输出中的非ASCII字符 `ñ` 是如何被 `ñ` 取代的。

5.17 将字节写入文本文件

问题

你想在文本模式打开的文件中写入原始的字节数据。

解决方案

将字节数据直接写入文件的缓冲区即可，例如：

```
>>> import sys
>>> sys.stdout.write(b'Hello\n')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>> sys.stdout.buffer.write(b'Hello\n')
Hello
5
>>>
```

类似的，能够通过读取文本文件的 `buffer` 属性来读取二进制数据。

讨论

I/O系统以层级结构的形式构建而成。文本文件是通过在一个拥有缓冲的二进制模式文件上增加一个Unicode编码/解码层来创建。 `buffer` 属性指向对应的底层文件。如果你直接访问它的话就会绕过文本编码/解码层。

本小节例子展示的 `sys.stdout` 可能看起来有点特殊。默认情况下， `sys.stdout` 总是以文本模式打开的。但是如果你在写一个需要打印二进制数据到标准输出的脚本的话，你可以使用上面演示的技术来绕过文本编码层。

5.18 将文件描述符包装成文件对象

问题

你有一个对应于操作系统上一个已打开的I/O通道(比如文件、管道、套接字等)的整型文件描述符，你想将它包装成一个更高层的Python文件对象。

解决方案

一个文件描述符和一个打开的普通文件是不一样的。文件描述符仅仅是一个由操作系统指定的整数，用来指代某个系统的I/O通道。如果你碰巧有这么一个文件描述符，你可以通过使用 `open()` 函数来将其包装为一个Python的文件对象。你仅仅只需要使用这个整数值文件描述符作为第一个参数来代替文件名即可。例如：

```
# Open a low-level file descriptor
import os
fd = os.open('somefile.txt', os.O_WRONLY | os.O_CREAT)

# Turn into a proper file
f = open(fd, 'wt')
f.write('hello world\n')
f.close()
```

当高层的文件对象被关闭或者破坏的时候，底层的文件描述符也会被关闭。如果这个并不是你想要的结果，你可以给 `open()` 函数传递一个可选的 `closefd=False`。比如：

```
# Create a file object, but don't close underlying fd when done
f = open(fd, 'wt', closefd=False)
...
```

讨论

在Unix系统中，这种包装文件描述符的技术可以很方便的将一个类文件接口作用于一个以不同方式打开的I/O通道上，如管道、套接字等。举例来讲，下面是一个操作管道的例子：

```

from socket import socket, AF_INET, SOCK_STREAM

def echo_client(client_sock, addr):
    print('Got connection from', addr)

    # Make text-mode file wrappers for socket reading/writing
    client_in = open(client_sock.fileno(), 'rt', encoding='latin-1',
                     closefd=False)

    client_out = open(client_sock.fileno(), 'wt', encoding='latin-1',
                      closefd=False)

    # Echo lines back to the client using file I/O
    for line in client_in:
        client_out.write(line)
        client_out.flush()

    client_sock.close()

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        echo_client(client, addr)

```

需要重点强调的一点是，上面的例子仅仅是为了演示内置的 `open()` 函数的一个特性，并且也只适用于基于 **Unix** 的系统。如果你想将一个类文件接口作用在一个套接字并希望你的代码可以跨平台，请使用套接字对象的 `makefile()` 方法。但是如果不考虑可移植性的话，那上面的解决方案会比使用 `makefile()` 性能更好一点。

你也可以使用这种技术来构造一个别名，允许以不同于第一次打开文件的方式使用它。例如，下面演示如何创建一个文件对象，它允许你输出二进制数据到标准输出(通常以文本模式打开):

```

import sys
# Create a binary-mode file for stdout
bstdout = open(sys.stdout.fileno(), 'wb', closefd=False)
bstdout.write(b'Hello World\n')
bstdout.flush()

```

尽管可以将一个已存在的文件描述符包装成一个正常的文件对象，但是要注意的是并不是所有的文件模式都被支持，并且某些类型的文件描述符可能会有副作用(特别是涉及到错误处理、文件结尾条件等等的时候)。在不同的操作系统上这种行为也是不一样，特别的，上面的例子都不能在非 **Unix** 系统上运行。我说了这么多，意思就是让你充分测试自己的实现代码，确保它能按照期望工作。

5.19 创建临时文件和文件夹

问题

你需要在程序执行时创建一个临时文件或目录，并希望使用完之后可以自动销毁掉。

解决方案

`tempfile` 模块中有很多的函数可以完成这任务。为了创建一个匿名的临时文件，可以使用 `tempfile.TemporaryFile`：

```
from tempfile import TemporaryFile

with TemporaryFile('w+t') as f:
    # Read/write to the file
    f.write('Hello World\n')
    f.write('Testing\n')

    # Seek back to beginning and read the data
    f.seek(0)
    data = f.read()

# Temporary file is destroyed
```

或者，如果你喜欢，你还可以像这样使用临时文件：

```
f = TemporaryFile('w+t')
# Use the temporary file
...
f.close()
# File is destroyed
```

`TemporaryFile()` 的第一个参数是文件模式，通常来讲文本模式使用 `w+t`，二进制模式使用 `w+b`。这个模式同时支持读和写操作，在这里是很有用的，因为当你关闭文件去改变模式的时候，文件实际上已经不存在了。`TemporaryFile()` 另外还支持跟内置的 `open()` 函数一样的参数。比如：

```
with TemporaryFile('w+t', encoding='utf-8', errors='ignore') as f:
    ...
```

在大多数Unix系统上，通过 `TemporaryFile()` 创建的文件都是匿名的，甚至连目录都没有。如果你想打破这个限制，可以使用 `NamedTemporaryFile()` 来代替。比如：

```
from tempfile import NamedTemporaryFile

with NamedTemporaryFile('w+t') as f:
    print('filename is:', f.name)
    ...

# File automatically destroyed
```

这里，被打开文件的 `f.name` 属性包含了该临时文件的文件名。当你需要将文件名传递给其他代码来打开这个文件的时候，这个就很有用了。和 `TemporaryFile()` 一样，结果文件关闭时会被自动删除掉。如果你不想这么做，可以传递一个关键字参数 `delete=False` 即可。比如：

```
with NamedTemporaryFile('w+t', delete=False) as f:
    print('filename is:', f.name)
    ...
```

为了创建一个临时目录，可以使用 `tempfile.TemporaryDirectory()`。比如：

```
from tempfile import TemporaryDirectory

with TemporaryDirectory() as dirname:
    print('dirname is:', dirname)
    # Use the directory
    ...

# Directory and all contents destroyed
```

讨论

`TemporaryFile()`、`NamedTemporaryFile()` 和 `TemporaryDirectory()` 函数应该是处理临时文件目录的最简单的方式了，因为它们会自动处理所有的创建和清理步骤。在一个更低的级别，你可以使用 `mkstemp()` 和 `mkdtemp()` 来创建临时文件和目录。比如：

```
>>> import tempfile
>>> tempfile.mkstemp()
(3, '/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp7fefhv')
>>> tempfile.mkdtemp()
'/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp5wvcv6'
>>>
```

但是，这些函数并不会做进一步的管理了。例如，函数 `mkstemp()` 仅仅就返回一个原始的OS文件描述符，你需要自己将它转换为一个真正的文件对象。同样你还需要自己清理这些文件。

通常来讲，临时文件在系统默认的位置被创建，比如 `/var/tmp` 或类似的地方。为了获取真实的位置，可以使用 `tempfile.gettempdir()` 函数。比如：

```
>>> tempfile.gettempdir()
'/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-'
>>>
```

所有和临时文件相关的函数都允许你通过使用关键字参数 `prefix`、`suffix` 和 `dir` 来自定义目录以及命名规则。比如：

```
>>> f = NamedTemporaryFile(prefix='mytemp', suffix='.txt', dir='/tmp')
>>> f.name
'/tmp/mytemp8ee899.txt'
>>>
```

最后还有一点，尽可能以最安全的方式使用 `tempfile` 模块来创建临时文件。包括仅给当前用户授权访问以及在文件创建过程中采取措施避免竞态条件。要注意的是不同的平台可能会不一样。因此你最好阅读 [官方文档](#) 来了解更多的细节。

5.20 与串行端口的数据通信

问题

你想通过串行端口读写数据，典型场景就是和一些硬件设备打交道(比如一个机器人或传感器)。

解决方案

尽管你可以通过使用Python内置的I/O模块来完成这个任务，但对于串行通信最好的选择是使用 `pySerial` 包。这个包的使用非常简单，先安装 `pySerial`，使用类似下面这样的代码就能很容易的打开一个串行端口：

```
import serial
ser = serial.Serial('/dev/tty.usbmodem641', # Device name varies
                   baudrate=9600,
                   bytesize=8,
                   parity='N',
                   stopbits=1)
```

设备名对于不同的设备和操作系统是不一样的。比如，在Windows系统上，你可以使用0, 1等表示的一个设备来打开通信端口“COM0”和“COM1”。一旦端口打开，那就可以使用 `read()`，`readline()` 和 `write()` 函数读写数据了。例如：

```
ser.write(b'G1 X50 Y50\r\n')
resp = ser.readline()
```

大多数情况下，简单的串口通信从此变得十分简单。

讨论

尽管表面上看起来很简单，其实串口通信有时候也是挺麻烦的。推荐你使用第三方包如 `pySerial` 的一个原因是它提供了对高级特性的支持 (比如超时，控制流，缓冲区刷新，握手协议等等)。举个例子，如果你想启用 `RTS-CTS` 握手协议，你只需要给 `Serial()` 传递一个 `rtscts=True` 的参数即可。其官方文档非常完善，因此我在这里极力推荐这个包。

时刻记住所有涉及到串口的I/O都是二进制模式的。因此，确保你的代码使用的是字节而不是文本 (或有时候执行文本的编码/解码操作)。另外当你需要创建二进制编码的指令或数据包的时候，`struct` 模块也是非常有用的。

5.21 序列化Python对象

问题

你需要将一个Python对象序列化为一个字节流，以便将它保存到一个文件、存储到数据库或者通过网络传输它。

解决方案

对于序列化最普遍的做法就是使用 `pickle` 模块。为了将一个对象保存到一个文件中，可以这样做：

```
import pickle

data = ... # Some Python object
f = open('somefile', 'wb')
pickle.dump(data, f)
```

为了将一个对象转储为一个字符串，可以使用 `pickle.dumps()`：

```
s = pickle.dumps(data)
```

为了从字节流中恢复一个对象，使用 `pickle.load()` 或 `pickle.loads()` 函数。比如：

```
# Restore from a file
f = open('somefile', 'rb')
data = pickle.load(f)

# Restore from a string
data = pickle.loads(s)
```

讨论

对于大多数应用程序来讲，`dump()` 和 `load()` 函数的使用就是你有效使用 `pickle` 模块所需的全部了。它可适用于绝大部分Python数据类型和用户自定义类的对象实例。如果你碰到某个库可以让你在数据库中保存/恢复Python对象或者是通过网络传输对象的话，那么很有可能这个库的底层就使用了 `pickle` 模块。

`pickle` 是一种Python特有的自描述的数据编码。通过自描述，被序列化后的数据包含每个对象开始和结束以及它的类型信息。因此，你无需担心对象记录的定义，它总是能工作。举个例子，如果要处理多个对象，你可以这样做：


```
>>> import pickle
>>> f = open('somedata', 'wb')
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump('hello', f)
>>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
>>> f.close()
>>> f = open('somedata', 'rb')
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'hello'
>>> pickle.load(f)
{'Apple', 'Pear', 'Banana'}
>>>
```

你还能序列化函数，类，还有接口，但是结果数据仅仅将它们名称编码成对应的代码对象。例如：

```
>>> import math
>>> import pickle.
>>> pickle.dumps(math.cos)
b'\x80\x03cmath\ncos\nq\x00.'
>>>
```

当数据反序列化回来的时候，会先假定所有的源数据是可用的。模块、类和函数会自动按需导入进来。对于Python数据被不同机器上的解析器所共享的应用程序而言，数据的保存可能会有问题，因为所有的机器都必须访问同一个源代码。

注

千万不要对不信任的数据使用`pickle.load()`。
`pickle`在加载时有一个副作用就是它会自动加载相应模块并构造实例对象。
但是某个坏人如果知道`pickle`的工作原理，
他就可以创建一个恶意的数据导致Python执行随意指定的系统命令。
因此，一定要保证`pickle`只在相互之间可以认证对方的解析器的内部使用。

有些类型的对象是不能被序列化的。这些通常是那些依赖外部系统状态的对象，比如打开的文件，网络连接，线程，进程，栈帧等等。用户自定义类可以通过提供

`__getstate__()` 和 `__setstate__()` 方法来绕过这些限制。如果定义了这两个方

法，`pickle.dump()` 就会调用 `__getstate__()` 获取序列化的对象。类似

的，`__setstate__()` 在反序列化时被调用。为了演示这个工作原理，下面是一个在内部定义了一个线程但仍然可以序列化和反序列化的类：

```

# countdown.py
import time
import threading

class Countdown:
    def __init__(self, n):
        self.n = n
        self.thr = threading.Thread(target=self.run)
        self.thr.daemon = True
        self.thr.start()

    def run(self):
        while self.n > 0:
            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

    def __getstate__(self):
        return self.n

    def __setstate__(self, n):
        self.__init__(n)

```

试着运行下面的序列化试验代码：

```

>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...

>>> # After a few moments
>>> f = open('cstate.p', 'wb')
>>> import pickle
>>> pickle.dump(c, f)
>>> f.close()

```

然后退出Python解析器并重启后再试验下：

```

>>> f = open('cstate.p', 'rb')
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...

```

你可以看到线程又奇迹般的重生了，从你第一次序列化它的地方又恢复过来。

`pickle` 对于大型的数据结构比如使用 `array` 或 `numpy` 模块创建的二进制数组效率并不是一个高效的编码方式。如果你需要移动大量的数组数据，你最好是先在一个文件中将其保存为数组数据块或使用更高级的标准编码方式如HDF5(需要第三方库的支持)。

由于 `pickle` 是Python特有的并且附着在源码上，所有如果需要长期存储数据的时候不应该选用它。例如，如果源码变动了，你所有的存储数据可能会被破坏并且变得不可读取。坦白来讲，对于在数据库和存档文件中存储数据时，你最好使用更加标准的数据编码格式如XML，CSV或JSON。这些编码格式更标准，可以被不同的语言支持，并且也能很好的适应源码变更。

最后一点要注意的是 `pickle` 有大量的配置选项和一些棘手的问题。对于最常见的使用场景，你不需要去担心这个，但是如果你要在一个重要的程序中使用`pickle`去做序列化的话，最好去查阅一下 [官方文档](#)。

第六章：数据编码和处理

这一章主要讨论使用Python处理各种不同方式编码的数据，比如CSV文件，JSON，XML和二进制包装记录。和数据结构那一章不同的是，这章不会讨论特殊的算法问题，而是关注于怎样获取和存储这些格式的数据。

Contents:

6.1 读写CSV数据

问题

你想读写一个CSV格式的文件。

解决方案

对于大多数的CSV格式的数据读写问题，都可以使用 `csv` 库。例如：假设你在一个名叫 `stocks.csv` 文件中有一些股票市场数据，就像这样：

```
Symbol,Price,Date,Time,Change,Volume
"AA",39.48,"6/11/2007","9:36am",-0.18,181800
"AIG",71.38,"6/11/2007","9:36am",-0.15,195500
"AXP",62.58,"6/11/2007","9:36am",-0.46,935000
"BA",98.31,"6/11/2007","9:36am",+0.12,104800
"C",53.08,"6/11/2007","9:36am",-0.25,360900
"CAT",78.29,"6/11/2007","9:36am",-0.23,225400
```

下面向你展示如何将这数据读取为一个元组的序列:

```
import csv
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
        # Process row
        ...
```

在上面的代码中, `row` 会是一个元组。因此, 为了访问某个字段, 你需要使用下标, 如 `row[0]` 访问Symbol, `row[4]` 访问Change。

由于这种下标访问通常会引起混淆, 你可以考虑使用命名元组。例如:

```
from collections import namedtuple
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headings = next(f_csv)
    Row = namedtuple('Row', headings)
    for r in f_csv:
        row = Row(*r)
        # Process row
        ...
```

它允许你使用列名如 `row.Symbol` 和 `row.Change` 代替下标访问。需要注意的是这个只有在列名是合法的Python标识符的时候才生效。如果不是的话, 你可能需要修改下原始的列名(如将非标识符字符替换成下划线之类的)。

另外一个选择就是将数据读取到一个字典序列中去。可以这样做:

```
import csv
with open('stocks.csv') as f:
    f_csv = csv.DictReader(f)
    for row in f_csv:
        # process row
    ...
```

在这个版本中，你可以使用列名去访问每一行的数据了。比如，`row['Symbol']` 或者 `row['Change']`。

为了写入CSV数据，你仍然可以使用csv模块，不过这时候先创建一个 `writer` 对象。例如：

```
headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [('AA', 39.48, '6/11/2007', '9:36am', -0.18, 181800),
        ('AIG', 71.38, '6/11/2007', '9:36am', -0.15, 195500),
        ('AXP', 62.58, '6/11/2007', '9:36am', -0.46, 935000),
        ]

with open('stocks.csv', 'w') as f:
    f_csv = csv.writer(f)
    f_csv.writerow(headers)
    f_csv.writerows(rows)
```

如果你有一个字典序列的数据，可以像这样做：

```
headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [{'Symbol': 'AA', 'Price': 39.48, 'Date': '6/11/2007',
        'Time': '9:36am', 'Change': -0.18, 'Volume': 181800},
        {'Symbol': 'AIG', 'Price': 71.38, 'Date': '6/11/2007',
        'Time': '9:36am', 'Change': -0.15, 'Volume': 195500},
        {'Symbol': 'AXP', 'Price': 62.58, 'Date': '6/11/2007',
        'Time': '9:36am', 'Change': -0.46, 'Volume': 935000},
        ]

with open('stocks.csv', 'w') as f:
    f_csv = csv.DictWriter(f, headers)
    f_csv.writeheader()
    f_csv.writerows(rows)
```

讨论

你应该总是优先选择csv模块分割或解析CSV数据。例如，你可能会像编写类似下面这样的代码：

```
with open('stocks.csv') as f:
    for line in f:
        row = line.split(',')
        # process row
        ...
```

使用这种方式的一个缺点就是你仍然需要去处理一些棘手的细节问题。比如，如果某些字段值被引号包围，你不得不去除这些引号。另外，如果一个被引号包围的字段碰巧含有一个逗号，那么程序就会因为产生一个错误大小的行而出错。

默认情况下，`csv` 库可识别Microsoft Excel所使用的CSV编码规则。这或许也是最常见的形式，并且也会给你带来最好的兼容性。然而，如果你查看csv的文档，就会发现有很多种方法将它应用到其他编码格式上(如修改分割字符等)。例如，如果你想读取以tab分割的数据，可以这样做：

```
# Example of reading tab-separated values
with open('stock.tsv') as f:
    f_tsv = csv.reader(f, delimiter='\t')
    for row in f_tsv:
        # Process row
        ...
```

如果你正在读取CSV数据并将它们转换为命名元组，需要注意对列名进行合法性认证。例如，一个CSV格式文件有一个包含非法标识符的列头行，类似下面这样：

这样最终会导致在创建一个命名元组时产生一个 `ValueError` 异常而失败。为了解决这问题，你可能不得不先去修正列标题。例如，可以像下面这样在非法标识符上使用一个正则表达式替换：

```
import re
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headers = [ re.sub('[^a-zA-Z]', '_', h) for h in next(f_csv) ]
    Row = namedtuple('Row', headers)
    for r in f_csv:
        row = Row(*r)
        # Process row
        ...
```

还有重要的一点需要强调的是，`csv`产生的数据都是字符串类型的，它不会做任何其他类型的转换。如果你需要做这样的类型转换，你必须自己手动去实现。下面是一个在CSV数据上执行其他类型转换的例子：

```
col_types = [str, float, str, str, float, int]
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
        # Apply conversions to the row items
        row = tuple(convert(value) for convert, value in zip(col_types, row))
        ...
```

另外，下面是一个转换字典中特定字段的例子：

```
print('Reading as dicts with type conversion')
field_types = [ ('Price', float),
                ('Change', float),
                ('Volume', int) ]

with open('stocks.csv') as f:
    for row in csv.DictReader(f):
        row.update((key, conversion(row[key]))
                   for key, conversion in field_types)
        print(row)
```

通常来讲，你可能并不想过多去考虑这些转换问题。在实际情况中，CSV文件都或多或少有些缺失的数据，被破坏的数据以及其它一些让转换失败的问题。因此，除非你的数据确实有保障是准确无误的，否则你必须考虑这些问题(你可能需要增加合适的错误处理机制)。

最后，如果你读取CSV数据的目的是做数据分析和统计的话，你可能需要看一看 `Pandas` 包。`Pandas` 包含了一个非常方便的函数叫 `pandas.read_csv()`，它可以加载CSV数据到一个 `DataFrame` 对象中去。然后利用这个对象你就可以生成各种形式的统计、过滤数据以及执行其他高级操作了。在6.13小节中会有这样一个例子。

6.2 读写JSON数据

问题

你想读写JSON(JavaScript Object Notation)编码格式的数据。

解决方案

`json` 模块提供了一种很简单的方式来编码和解码JSON数据。其中两个主要的函数是 `json.dumps()` 和 `json.loads()`，要比其他序列化函数库如pickle的接口少得多。下面演示如何将一个Python数据结构转换为JSON：

```
import json

data = {
    'name' : 'ACME',
    'shares' : 100,
    'price' : 542.23
}

json_str = json.dumps(data)
```

下面演示如何将一个JSON编码的字符串转换回一个Python数据结构：

```
data = json.loads(json_str)
```

如果你要处理的是文件而不是字符串，你可以使用 `json.dump()` 和 `json.load()` 来编码和解码JSON数据。例如：

```
# Writing JSON data
with open('data.json', 'w') as f:
    json.dump(data, f)

# Reading data back
with open('data.json', 'r') as f:
    data = json.load(f)
```

讨论

JSON编码支持的基本数据类型为 `None`，`bool`，`int`，`float` 和 `str`，以及包含这些类型数据的lists，tuples和dictionaries。对于dictionaries，keys需要是字符串类型(字典中任何非字符串类型的key在编码时会先转换为字符串)。为了遵循JSON规范，你应该只编码Python的lists和dictionaries。而且，在web应用程序中，顶层对象被编码为一个字典是一个标准做法。

JSON编码的格式对于Python语法而已几乎是完全一样的，除了一些小的差异之外。比如，`True`会被映射为`true`，`False`被映射为`false`，而`None`会被映射为`null`。下面是一个例子，演示了编码后的字符串效果：

```
>>> json.dumps(False)
'false'
>>> d = {'a': True,
...      'b': 'Hello',
...      'c': None}
>>> json.dumps(d)
'{"b": "Hello", "c": null, "a": true}'
>>>
```

如果你试着去检查JSON解码后的数据，你通常很难通过简单的打印来确定它的结构，特别是当数据的嵌套结构层次很深或者包含大量的字段时。为了解决这个问题，可以考虑使用`pprint`模块的 `pprint()` 函数来代替普通的 `print()` 函数。它会按照`key`的字母顺序并以一种更加美观的方式输出。下面是一个演示如何漂亮的打印输出Twitter上搜索结果的例子：

```

>>> from urllib.request import urlopen
>>> import json
>>> u = urlopen('http://search.twitter.com/search.json?q=python&rpp=5')
>>> resp = json.loads(u.read().decode('utf-8'))
>>> from pprint import pprint
>>> pprint(resp)
{'completed_in': 0.074,
 'max_id': 264043230692245504,
 'max_id_str': '264043230692245504',
 'next_page': '?page=2&max_id=264043230692245504&q=python&rpp=5',
 'page': 1,
 'query': 'python',
 'refresh_url': '?since_id=264043230692245504&q=python',
 'results': [{ 'created_at': 'Thu, 01 Nov 2012 16:36:26 +0000',
                'from_user': ...
              },
             { 'created_at': 'Thu, 01 Nov 2012 16:36:14 +0000',
                'from_user': ...
              },
             { 'created_at': 'Thu, 01 Nov 2012 16:36:13 +0000',
                'from_user': ...
              },
             { 'created_at': 'Thu, 01 Nov 2012 16:36:07 +0000',
                'from_user': ...
              },
             { 'created_at': 'Thu, 01 Nov 2012 16:36:04 +0000',
                'from_user': ...
              }
            ],
 'results_per_page': 5,
 'since_id': 0,
 'since_id_str': '0'}
>>>

```

一般来讲，JSON解码会根据提供的数据创建dicts或lists。如果你想要创建其他类型的对象，可以给 `json.loads()` 传递 `object_pairs_hook` 或 `object_hook` 参数。例如，下面是演示如何解码JSON数据并在一个OrderedDict中保留其顺序的例子：

```

>>> s = '{"name": "ACME", "shares": 50, "price": 490.1}'
>>> from collections import OrderedDict
>>> data = json.loads(s, object_pairs_hook=OrderedDict)
>>> data
OrderedDict([('name', 'ACME'), ('shares', 50), ('price', 490.1)])
>>>

```

下面是如何将一个JSON字典转换为一个Python对象例子：

```
>>> class JSONObject:
...     def __init__(self, d):
...         self.__dict__ = d
...
>>>
>>> data = json.loads(s, object_hook=JSONObject)
>>> data.name
'ACME'
>>> data.shares
50
>>> data.price
490.1
>>>
```

最后一个例子中，JSON解码后的字典作为一个单个参数传递给 `__init__()`。然后，你就可以随心所欲的使用它了，比如作为一个实例字典来直接使用它。

在编码JSON的时候，还有一些选项很有用。如果你想获得漂亮的格式化字符串后输出，可以使用 `json.dumps()` 的 `indent` 参数。它会使得输出和 `pprint()` 函数效果类似。比如：

```
>>> print(json.dumps(data))
{"price": 542.23, "name": "ACME", "shares": 100}
>>> print(json.dumps(data, indent=4))
{
    "price": 542.23,
    "name": "ACME",
    "shares": 100
}
>>>
```

对象实例通常并不是JSON可序列化的。例如：

```

>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> p = Point(2, 3)
>>> json.dumps(p)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/json/__init__.py", line 226, in dumps
    return _default_encoder.encode(obj)
  File "/usr/local/lib/python3.3/json/encoder.py", line 187, in encode
    chunks = self.iterencode(o, _one_shot=True)
  File "/usr/local/lib/python3.3/json/encoder.py", line 245, in iterencode
    return _iterencode(o, 0)
  File "/usr/local/lib/python3.3/json/encoder.py", line 169, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: <__main__.Point object at 0x1006f2650> is not JSON serializable
>>>

```

如果你想序列化对象实例，你可以提供一个函数，它的输入是一个实例，返回一个可序列化的字典。例如：

```

def serialize_instance(obj):
    d = { '__classname__' : type(obj).__name__ }
    d.update(vars(obj))
    return d

```

如果你想反过来获取这个实例，可以这样做：

```

# Dictionary mapping names to known classes
classes = {
    'Point' : Point
}

def unserialize_object(d):
    clsname = d.pop('__classname__', None)
    if clsname:
        cls = classes[clsname]
        obj = cls.__new__(cls) # Make instance without calling __init__
        for key, value in d.items():
            setattr(obj, key, value)
        return obj
    else:
        return d

```

下面是如何使用这些函数的例子：

```
>>> p = Point(2,3)
>>> s = json.dumps(p, default=serialize_instance)
>>> s
'{"__classname__": "Point", "y": 3, "x": 2}'
>>> a = json.loads(s, object_hook=unserialize_object)
>>> a
<__main__.Point object at 0x1017577d0>
>>> a.x
2
>>> a.y
3
>>>
```

`json` 模块还有很多其他选项来控制更低级别的数字、特殊值如NaN等的解析。可以参考官方文档获取更多细节。

6.3 解析简单的XML数据

问题

你想从一个简单的XML文档中提取数据。

解决方案

可以使用 `xml.etree.ElementTree` 模块从简单的XML文档中提取数据。为了演示，假设你想解析Planet Python上的RSS源。下面是相应的代码：

```
from urllib.request import urlopen
from xml.etree.ElementTree import parse

# Download the RSS feed and parse it
u = urlopen('http://planet.python.org/rss20.xml')
doc = parse(u)

# Extract and output tags of interest
for item in doc.iterfind('channel/item'):
    title = item.findtext('title')
    date = item.findtext('pubDate')
    link = item.findtext('link')

    print(title)
    print(date)
    print(link)
    print()
```

运行上面的代码，输出结果类似这样：

```
Steve Holden: Python for Data Analysis
Mon, 19 Nov 2012 02:13:51 +0000
http://holdenweb.blogspot.com/2012/11/python-for-data-analysis.html

Vasudev Ram: The Python Data model (for v2 and v3)
Sun, 18 Nov 2012 22:06:47 +0000
http://jugad2.blogspot.com/2012/11/the-python-data-model.html

Python Diary: Been playing around with Object Databases
Sun, 18 Nov 2012 20:40:29 +0000
http://www.pythondiary.com/blog/Nov.18,2012/been-...-object-databases.html

Vasudev Ram: Wakari, Scientific Python in the cloud
Sun, 18 Nov 2012 20:19:41 +0000
http://jugad2.blogspot.com/2012/11/wakari-scientific-python-in-cloud.html

Jesse Jiryu Davis: Toro: synchronization primitives for Tornado coroutines
Sun, 18 Nov 2012 20:17:49 +0000
http://feedproxy.google.com/~r/EmptysquarePython/~3/_D0ZT2Kd0hQ/
```

很显然，如果你想做进一步的处理，你需要替换 `print()` 语句来完成其他有趣的事。

讨论

在很多应用程序中处理XML编码格式的数据是很常见的。不仅因为XML在Internet上面已经被广泛应用于数据交换，同时它也是一种存储应用程序数据的常用格式(比如字处理，音乐库等)。接下来的讨论会先假定读者已经对XML基础比较熟悉了。

在很多情况下，当使用XML来仅仅存储数据的时候，对应的文档结构非常紧凑并且直观。例如，上面例子中的RSS订阅源类似于下面的格式：

```

<?xml version="1.0"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel>
    <title>Planet Python</title>
    <link>http://planet.python.org/</link>
    <language>en</language>
    <description>Planet Python - http://planet.python.org/</description>
    <item>
      <title>Steve Holden: Python for Data Analysis</title>
      <guid>http://holdenweb.blogspot.com/...-data-analysis.html</guid>
      <link>http://holdenweb.blogspot.com/...-data-analysis.html</link>
      <description>...</description>
      <pubDate>Mon, 19 Nov 2012 02:13:51 +0000</pubDate>
    </item>
    <item>
      <title>Vasudev Ram: The Python Data model (for v2 and v3)</title>
      <guid>http://jugad2.blogspot.com/...-data-model.html</guid>
      <link>http://jugad2.blogspot.com/...-data-model.html</link>
      <description>...</description>
      <pubDate>Sun, 18 Nov 2012 22:06:47 +0000</pubDate>
    </item>
    <item>
      <title>Python Diary: Been playing around with Object Databases</title>
      <guid>http://www.pythondiary.com/...-object-databases.html</guid>
      <link>http://www.pythondiary.com/...-object-databases.html</link>
      <description>...</description>
      <pubDate>Sun, 18 Nov 2012 20:40:29 +0000</pubDate>
    </item>
    ...
  </channel>
</rss>

```

`xml.etree.ElementTree.parse()` 函数解析整个XML文档并将其转换成一个文档对象。然后，你就能使用 `find()`、`iterfind()` 和 `findtext()` 等方法来搜索特定的XML元素了。这些函数的参数就是某个指定的标签名，例如 `channel/item` 或 `title`。

每次指定某个标签时，你需要遍历整个文档结构。每次搜索操作会从一个起始元素开始进行。同样，每次操作所指定的标签名也是起始元素的相对路径。例如，执行 `doc.iterfind('channel/item')` 来搜索所有在 `channel` 元素下面的 `item` 元素。`doc` 代表文档的最顶层(也就是第一级的 `rss` 元素)。然后接下来的调用 `item.findtext()` 会从已找到的 `item` 元素位置开始搜索。

`ElementTree` 模块中的每个元素有一些重要的属性和方法，在解析的时候非常有用。`tag` 属性包含了标签的名字，`text` 属性包含了内部的文本，而 `get()` 方法能获取属性值。例如：

```
>>> doc
<xml.etree.ElementTree.ElementTree object at 0x101339510>
>>> e = doc.find('channel/title')
>>> e
<Element 'title' at 0x10135b310>
>>> e.tag
'title'
>>> e.text
'Planet Python'
>>> e.get('some_attribute')
>>>
```

有一点要强调的是 `xml.etree.ElementTree` 并不是XML解析的唯一方法。对于更高级的应用程序，你需要考虑使用 `lxml`。它使用了和ElementTree同样的编程接口，因此上面的例子同样也适用于lxml。你只需要将刚开始的import语句换成 `from lxml.etree import parse` 就行了。`lxml` 完全遵循XML标准，并且速度也非常快，同时还支持验证，XSLT，和XPath等特性。

6.4 增量式解析大型XML文件

问题

你想使用尽可能少的内存从一个超大的XML文档中提取数据。

解决方案

任何时候只要你遇到增量式的数据处理时，第一时间就应该想到迭代器和生成器。下面是一个很简单的函数，只使用很少的内存就能增量式的处理一个大型XML文件：


```

from xml.etree.ElementTree import iterparse

def parse_and_remove(filename, path):
    path_parts = path.split('/')
    doc = iterparse(filename, ('start', 'end'))
    # Skip the root element
    next(doc)

    tag_stack = []
    elem_stack = []
    for event, elem in doc:
        if event == 'start':
            tag_stack.append(elem.tag)
            elem_stack.append(elem)
        elif event == 'end':
            if tag_stack == path_parts:
                yield elem
                elem_stack[-2].remove(elem)
            try:
                tag_stack.pop()
                elem_stack.pop()
            except IndexError:
                pass

```

为了测试这个函数，你需要先有一个大型的XML文件。通常你可以在政府网站或公共数据网站上找到这样的文件。例如，你可以下载XML格式的芝加哥城市道路坑洼数据库。在写这本书的时候，下载文件已经包含超过100,000行数据，编码格式类似于下面这样：

假设你想写一个脚本来按照坑洼报告数量排列邮编号码。你可以像这样做：

```

from xml.etree.ElementTree import parse
from collections import Counter

potholes_by_zip = Counter()

doc = parse('potholes.xml')
for pothole in doc.iterfind('row/row'):
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)

```

这个脚本唯一的问题是它会先将整个XML文件加载到内存中然后解析。在我的机器上，为了运行这个程序需要用到450MB左右的内存空间。如果使用如下代码，程序只需要修改一点点：

```
from collections import Counter

potholes_by_zip = Counter()

data = parse_and_remove('potholes.xml', 'row/row')
for pothole in data:
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

结果是：这个版本的代码运行时只需要7MB的内存-大大节约了内存资源。

讨论

这一节的技术会依赖 `ElementTree` 模块中的两个核心功能。第一，`iterparse()` 方法允许对XML文档进行增量操作。使用时，你需要提供文件名和一个包含下面一种或多种类型的事件列表：`start`，`end`，`start-ns` 和 `end-ns`。由 `iterparse()` 创建的迭代器会产生形如 `(event, elem)` 的元组，其中 `event` 是上述事件列表中的某一个，而 `elem` 是相应的XML元素。例如：

```
>>> data = iterparse('potholes.xml', ('start', 'end'))
>>> next(data)
('start', <Element 'response' at 0x100771d60>)
>>> next(data)
('start', <Element 'row' at 0x100771e68>)
>>> next(data)
('start', <Element 'row' at 0x100771fc8>)
>>> next(data)
('start', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('end', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('start', <Element 'status' at 0x1006a7f18>)
>>> next(data)
('end', <Element 'status' at 0x1006a7f18>)
>>>
```

`start` 事件在某个元素第一次被创建并且还没有被插入其他数据(如子元素)时被创建。而 `end` 事件在某个元素已经完成时被创建。尽管没有在例子中演示，`start-ns` 和 `end-ns` 事件被用来处理XML文档命名空间的声明。

这节例子中，`start` 和 `end` 事件被用来管理元素和标签栈。栈代表了文档被解析时的层次结构，还被用来判断某个元素是否匹配传给函数 `parse_and_remove()` 的路径。如果匹配，就利用 `yield` 语句向调用者返回这个元素。

在 `yield` 之后的下面这个语句才是使得程序占用极少内存的 `ElementTree` 的核心特性：

```
elem_stack[-2].remove(elem)
```

这个语句使得之前由 `yield` 产生的元素从它的父节点中删除掉。假设已经没有其它的地方引用这个元素了，那么这个元素就被销毁并回收内存。

对节点的迭代式解析和删除的最终效果就是一个在文档上高效的增量式清扫过程。文档树结构从始至终没被完整的创建过。尽管如此，还是能通过上述简单的方式来处理这个 XML 数据。

这种方案的主要缺陷就是它的运行性能了。我自己测试的结果是，读取整个文档到内存中的版本的运行速度差不多是增量式处理版本的两倍快。但是它却使用了超过后者 60 倍的内存。因此，如果你更关心内存使用量的话，那么增量式的版本完胜。

6.5 将字典转换为 XML

问题

你想使用一个 Python 字典存储数据，并将它转换成 XML 格式。

解决方案

尽管 `xml.etree.ElementTree` 库通常用来做解析工作，其实它也可以创建 XML 文档。例如，考虑如下这个函数：

```
from xml.etree.ElementTree import Element

def dict_to_xml(tag, d):
    ...
    Turn a simple dict of key/value pairs into XML
    ...
    elem = Element(tag)
    for key, val in d.items():
        child = Element(key)
        child.text = str(val)
        elem.append(child)
    return elem
```

下面是一个使用例子：

```
>>> s = { 'name': 'GOOG', 'shares': 100, 'price':490.1 }
>>> e = dict_to_xml('stock', s)
>>> e
<Element 'stock' at 0x1004b64c8>
>>>
```

转换结果是一个 `Element` 实例。对于I/O操作，使用 `xml.etree.ElementTree` 中的 `tostring()` 函数很容易就能将它转换成一个字节字符串。例如：

```
>>> from xml.etree.ElementTree import tostring
>>> tostring(e)
b'<stock><price>490.1</price><shares>100</shares><name>GOOG</name></stock>'
```

如果你想给某个元素添加属性值，可以使用 `set()` 方法：

```
>>> e.set('_id', '1234')
>>> tostring(e)
b'<stock _id="1234"><price>490.1</price><shares>100</shares><name>GOOG</name></stock>'
```

如果你还想保持元素的顺序，可以考虑构造一个 `OrderedDict` 来代替一个普通的字典。请参考1.7小节。

讨论

当创建XML的时候，你被限制只能构造字符串类型的值。例如：

```
def dict_to_xml_str(tag, d):
    """
    Turn a simple dict of key/value pairs into XML
    """
    parts = ['<{}>'.format(tag)]
    for key, val in d.items():
        parts.append('<{0}>{1}</{0}>'.format(key, val))
    parts.append('</{}>'.format(tag))
    return ''.join(parts)
```

问题是如果你手动的去构造的时候可能会碰到一些麻烦。例如，当字典的值中包含一些特殊字符的时候会怎样呢？

```
>>> d = { 'name' : '<spam>' }

>>> # String creation
>>> dict_to_xml_str('item', d)
'<item><name><spam></name></item>'

>>> # Proper XML creation
>>> e = dict_to_xml('item', d)
>>> tostring(e)
b'<item><name>&lt;spam&gt;</name></item>'
>>>
```

注意到程序的后面那个例子中，字符‘<’和‘>’被替换成了 `<` 和 `>`

下面仅供参考，如果你需要手动去转换这些字符，可以使用 `xml.sax.saxutils` 中的 `escape()` 和 `unescape()` 函数。例如：

```
>>> from xml.sax.saxutils import escape, unescape
>>> escape('<spam>')
'&lt;spam&gt;'
>>> unescape(_)
'<spam>'
>>>
```

除了能创建正确的输出外，还有另外一个原因推荐你创建 `Element` 实例而不是字符串，那就是使用字符串组合构造一个更大的文档并不是那么容易。而 `Element` 实例可以不用考虑解析XML文本的情况下通过多种方式被处理。也就是说，你可以在一个高级数据结构上完成你所有的操作，并在最后以字符串的形式将其输出。

6.6 解析和修改XML

问题

你想读取一个XML文档，对它做一些修改，然后将结果写回XML文档。

解决方案

使用 `xml.etree.ElementTree` 模块可以很容易的处理这些任务。第一步是以通常的方式来解析这个文档。例如，假设你有一个名为 `pred.xml` 的文档，类似下面这样：

下面是一个利用 `ElementTree` 来读取这个文档并对它做一些修改的例子：

```
>>> from xml.etree.ElementTree import parse, Element
>>> doc = parse('pred.xml')
>>> root = doc.getroot()
>>> root
<Element 'stop' at 0x100770cb0>

>>> # Remove a few elements
>>> root.remove(root.find('sri'))
>>> root.remove(root.find('cr'))
>>> # Insert a new element after <nm>...</nm>
>>> root.getchildren().index(root.find('nm'))
1
>>> e = Element('spam')
>>> e.text = 'This is a test'
>>> root.insert(2, e)

>>> # Write back to a file
>>> doc.write('newpred.xml', xml_declaration=True)
>>>
```

处理结果是一个像下面这样新的XML文件：

讨论

修改一个XML文档结构是很容易的，但是你必须牢记的是所有的修改都是针对父节点元素，将它作为一个列表来处理。例如，如果你删除某个元素，通过调用父节点的 `remove()` 方法从它的直接父节点中删除。如果你插入或增加新的元素，你同样使用父节点元素的 `insert()` 和 `append()` 方法。还能对元素使用索引和切片操作，比如 `element[i]` 或 `element[i:j]`

如果你需要创建新的元素，可以使用本节方案中演示的 `Element` 类。我们在6.5小节已经详细讨论过了。

6.7 利用命名空间解析XML文档

问题

你想解析某个XML文档，文档中使用了XML命名空间。

解决方案

考虑下面这个使用了命名空间的文档：

如果你解析这个文档并执行普通的查询，你会发现这个并不是那么容易，因为所有步骤都变得相当的繁琐。

```
>>> # Some queries that work
>>> doc.findtext('author')
'David Beazley'
>>> doc.find('content')
<Element 'content' at 0x100776ec0>
>>> # A query involving a namespace (doesn't work)
>>> doc.find('content/html')
>>> # Works if fully qualified
>>> doc.find('content/{http://www.w3.org/1999/xhtml}html')
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> # Doesn't work
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/head/title')
>>> # Fully qualified
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/'
... '{http://www.w3.org/1999/xhtml}head/{http://www.w3.org/1999/xhtml}title')
'Hello World'
>>>
```

你可以通过将命名空间处理逻辑包装为一个工具类来简化这个过程：

```

class XMLNamespaces:
    def __init__(self, **kwargs):
        self.namespaces = {}
        for name, uri in kwargs.items():
            self.register(name, uri)
    def register(self, name, uri):
        self.namespaces[name] = '{'+uri+'}'
    def __call__(self, path):
        return path.format_map(self.namespaces)

```

通过下面的方式使用这个类：

```

>>> ns = XMLNamespaces(html='http://www.w3.org/1999/xhtml')
>>> doc.find(ns('content/{html}html'))
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> doc.findtext(ns('content/{html}html/{html}head/{html}title'))
'Hello World'
>>>

```

讨论

解析含有命名空间的XML文档会比较繁琐。上面的 `XMLNamespaces` 仅仅是允许你使用缩略名代替完整的URI将其变得稍微简洁一点。

很不幸的是，在基本的 `ElementTree` 解析中没有任何途径获取命名空间的信息。但是，如果你使用 `iterparse()` 函数的话就可以获取更多关于命名空间处理范围的信息。例如：

```

>>> from xml.etree.ElementTree import iterparse
>>> for evt, elem in iterparse('ns2.xml', ('end', 'start-ns', 'end-ns')):
...     print(evt, elem)
...
end <Element 'author' at 0x10110de10>
start-ns ('', 'http://www.w3.org/1999/xhtml')
end <Element '{http://www.w3.org/1999/xhtml}title' at 0x1011131b0>
end <Element '{http://www.w3.org/1999/xhtml}head' at 0x1011130a8>
end <Element '{http://www.w3.org/1999/xhtml}h1' at 0x101113310>
end <Element '{http://www.w3.org/1999/xhtml}body' at 0x101113260>
end <Element '{http://www.w3.org/1999/xhtml}html' at 0x10110df70>
end-ns None
end <Element 'content' at 0x10110de68>
end <Element 'top' at 0x10110dd60>
>>> elem # This is the topmost element
<Element 'top' at 0x10110dd60>
>>>

```


最后一点，如果你要处理的XML文本除了要使用到其他高级XML特性外，还要使用到命名空间，建议你最好是使用 `lxml` 函数库来代替 `ElementTree`。例如，`lxml` 对利用DTD验证文档、更好的XPath支持和一些其他高级XML特性等都提供了更好的支持。这一小节其实只是教你如何让XML解析稍微简单一点。

6.8 与关系型数据库的交互

问题

你想在关系型数据库中查询、增加或删除记录。

解决方案

Python中表示多行数据的标准方式是一个由元组构成的序列。例如：

```
stocks = [  
    ('GOOG', 100, 490.1),  
    ('AAPL', 50, 545.75),  
    ('FB', 150, 7.45),  
    ('HPQ', 75, 33.2),  
]
```

依据PEP249，通过这种形式提供数据，可以很容易的使用Python标准数据库API和关系型数据库进行交互。所有数据库上的操作都通过SQL查询语句来完成。每一行输入输出数据用一个元组来表示。

为了演示说明，你可以使用Python标准库中的 `sqlite3` 模块。如果你使用的是一个不同的数据库(比如MySQL、Postgresql或者ODBC)，还得安装相应的第三方模块来提供支持。不过相应的编程接口几乎都是一样的，除了一点点细微差别外。

第一步是连接到数据库。通常你要执行 `connect()` 函数，给它提供一些数据库名、主机、用户名、密码和其他必要的一些参数。例如：

```
>>> import sqlite3  
>>> db = sqlite3.connect('database.db')  
>>>
```

为了处理数据，下一步你需要创建一个游标。一旦你有了游标，那么你就可以执行SQL查询语句了。比如：

```
>>> c = db.cursor()
>>> c.execute('create table portfolio (symbol text, shares integer, price real)')
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

为了向数据库表中插入多条记录，使用类似下面这样的语句：

```
>>> c.executemany('insert into portfolio values (?, ?, ?)', stocks)
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

为了执行某个查询，使用像下面这样的语句：

```
>>> for row in db.execute('select * from portfolio'):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
('FB', 150, 7.45)
('HPQ', 75, 33.2)
>>>
```

如果你想接受用户输入作为参数来执行查询操作，必须确保你使用下面这样的占位符`?`来进行引用参数：

```
>>> min_price = 100
>>> for row in db.execute('select * from portfolio where price >= ?',
...                        (min_price,)):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
>>>
```

讨论

在比较低的级别上和数据库交互是非常简单的。你只需提供SQL语句并调用相应的模块就可以更新或提取数据了。虽说如此，还是有一些比较棘手的细节问题需要你逐个列出去解决。

一个难点是数据库中的数据和Python类型直接的映射。对于日期类型，通常可以使用 `datetime` 模块中的 `datetime` 实例，或者可能是 `time` 模块中的系统时间戳。对于数字类型，特别是使用到小数的金融数据，可以用 `decimal` 模块中的 `Decimal` 实例来表示。不幸的是，对于不同的数据库而言具体映射规则是不一样的，你必须参考相应的文档。

另外一个更加复杂的问题就是SQL语句字符串的构造。你千万不要使用Python字符串格式化操作符(如%)或者 `.format()` 方法来创建这样的字符串。如果传递给这些格式化操作符的值来自于用户的输入，那么你的程序就很有可能遭受SQL注入攻击(参考 <http://xkcd.com/327>)。查询语句中的通配符 `?` 指示后台数据库使用它自己的字符串替换机制，这样更加的安全。

不幸的是，不同的数据库后台对于通配符的使用是不一样的。大部分模块使用 `?` 或 `%s`，还有其他一些使用了不同的符号，比如:0或:1来指示参数。同样的，你还是得去参考你使用的数据库模块相应的文档。一个数据库模块的 `paramstyle` 属性包含了参数引用风格的信息。

对于简单的数据库数据的读写问题，使用数据库API通常非常简单。如果你要处理更加复杂的问题，建议你使用更加高级的接口，比如一个对象关系映射ORM所提供的接口。类似 `SQLAlchemy` 这样的库允许你使用Python类来表示一个数据库表，并且能在隐藏底层SQL的情况下实现各种数据库的操作。

6.9 编码和解码十六进制数

问题

你想将一个十六进制字符串解码成一个字节字符串或者将一个字节字符串编码成一个十六进制字符串。

解决方案

如果你只是简单的解码或编码一个十六进制的原始字符串，可以使用 `binascii` 模块。例如：

```
>>> # Initial byte string
>>> s = b'hello'
>>> # Encode as hex
>>> import binascii
>>> h = binascii.b2a_hex(s)
>>> h
b'68656c6c6f'
>>> # Decode back to bytes
>>> binascii.a2b_hex(h)
b'hello'
>>>
```

类似的功能同样可以在 `base64` 模块中找到。例如：

```
>>> import base64
>>> h = base64.b16encode(s)
>>> h
b'68656C6C6F'
>>> base64.b16decode(h)
b'hello'
>>>
```

讨论

大部分情况下，通过使用上述的函数来转换十六进制是很简单的。上面两种技术的主要不同在于大小写的处理。函数 `base64.b16decode()` 和 `base64.b16encode()` 只能操作大写形式的十六进制字母，而 `binascii` 模块中的函数大小写都能处理。

还有一点需要注意的是编码函数所产生的输出总是一个字节字符串。如果想强制以 `Unicode` 形式输出，你需要增加一个额外的界面步骤。例如：

```
>>> h = base64.b16encode(s)
>>> print(h)
b'68656C6C6F'
>>> print(h.decode('ascii'))
68656C6C6F
>>>
```

在解码十六进制数时，函数 `b16decode()` 和 `a2b_hex()` 可以接受字节或 `unicode` 字符串。但是，`unicode` 字符串必须仅仅只包含 `ASCII` 编码的十六进制数。

6.10 编码解码Base64数据

问题

你需要使用Base64格式解码或编码二进制数据。

解决方案

`base64` 模块中有两个函数 `b64encode()` and `b64decode()` 可以帮你解决这个问题。例如;

```
>>> # Some byte data
>>> s = b'hello'
>>> import base64

>>> # Encode as Base64
>>> a = base64.b64encode(s)
>>> a
b'aGVsbG8='

>>> # Decode from Base64
>>> base64.b64decode(a)
b'hello'
>>>
```

讨论

Base64编码仅仅用于面向字节的数据比如字节字符串和字节数组。此外，编码处理的输出结果总是一个字节字符串。如果你想混合使用Base64编码的数据和Unicode文本，你必须添加一个额外的解码步骤。例如：

```
>>> a = base64.b64encode(s).decode('ascii')
>>> a
'aGVsbG8='
>>>
```

当解码Base64的时候，字节字符串和Unicode文本都可以作为参数。但是，Unicode字符串只能包含ASCII字符。

6.11 读写二进制数组数据

问题

你想读写一个二进制数组的结构化数据到Python元组中。

解决方案

可以使用 `struct` 模块处理二进制数据。下面是一段示例代码将一个Python元组列表写入一个二进制文件，并使用 `struct` 将每个元组编码为一个结构体。

```
from struct import Struct
def write_records(records, format, f):
    """
    Write a sequence of tuples to a binary file of structures.
    """
    record_struct = Struct(format)
    for r in records:
        f.write(record_struct.pack(*r))

# Example
if __name__ == '__main__':
    records = [ (1, 2.3, 4.5),
                (6, 7.8, 9.0),
                (12, 13.4, 56.7) ]
    with open('data.b', 'wb') as f:
        write_records(records, '<idd', f)
```

有很多种方法来读取这个文件并返回一个元组列表。首先，如果你打算以块的形式增量读取文件，你可以这样做：

```
from struct import Struct

def read_records(format, f):
    record_struct = Struct(format)
    chunks = iter(lambda: f.read(record_struct.size), b'')
    return (record_struct.unpack(chunk) for chunk in chunks)

# Example
if __name__ == '__main__':
    with open('data.b', 'rb') as f:
        for rec in read_records('<idd', f):
            # Process rec
            ...
```

如果你想将整个文件一次性读取到一个字节字符串中，然后在分片解析。那么你可以这样做：

```
from struct import Struct

def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack_from(data, offset)
            for offset in range(0, len(data), record_struct.size))

# Example
if __name__ == '__main__':
    with open('data.b', 'rb') as f:
        data = f.read()
    for rec in unpack_records('<idd', data):
        # Process rec
        ...
```

两种情况下的结果都是一个可返回用来创建该文件的原始元组的可迭代对象。

讨论

对于需要编码和解码二进制数据的程序而言，通常会使用 `struct` 模块。为了声明一个新的结构体，只需要像这样创建一个 `Struct` 实例即可：

```
# Little endian 32-bit integer, two double precision floats
record_struct = Struct('<idd')
```

结构体通常会使用一些结构码值 `i`, `d`, `f` 等 [参考 [Python 文档](#)]。这些代码分别代表某个特定的二进制数据类型如 32 位整数，64 位浮点数，32 位浮点数等。第一个字符 `<` 指定了字节顺序。在这个例子中，它表示“低位在前”。更改这个字符为 `>` 表示高位在前，或者是 `!` 表示网络字节顺序。

产生的 `Struct` 实例有很多属性和方法用来操作相应类型的结构。`size` 属性包含了结构的字节数，这在 I/O 操作时非常有用。`pack()` 和 `unpack()` 方法被用来打包和解包数据。比如：

```

>>> from struct import Struct
>>> record_struct = Struct('<idd')
>>> record_struct.size
20
>>> record_struct.pack(1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x03@'
>>> record_struct.unpack(_)
(1, 2.0, 3.0)
>>>

```

有时候你还会看到 `pack()` 和 `unpack()` 操作以模块级别函数被调用，类似下面这样：

```

>>> import struct
>>> struct.pack('<idd', 1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x03@'
>>> struct.unpack('<idd', _)
(1, 2.0, 3.0)
>>>

```

这样可以工作，但是感觉没有实例方法那么优雅，特别是在你代码中同样的结构出现在多个地方的时候。通过创建一个 `Struct` 实例，格式代码只会指定一次并且所有的操作被集中处理。这样一来代码维护就变得更加简单了(因为你只需要改变一处代码即可)。

读取二进制结构的代码要用到一些非常有趣而优美的编程技巧。在函数 `read_records` 中，`iter()` 被用来创建一个返回固定大小数据块的迭代器，参考5.8小节。这个迭代器会不断的调用一个用户提供的可调用对象(比如 `lambda: f.read(record_struct.size)`)，直到它返回一个特殊的值(如b"")，这时候迭代停止。例如：

```

>>> f = open('data.b', 'rb')
>>> chunks = iter(lambda: f.read(20), b'')
>>> chunks
<callable_iterator object at 0x10069e6d0>
>>> for chk in chunks:
...     print(chk)
...
b'\x01\x00\x00\x00ffffff\x02@\x00\x00\x00\x00\x00\x00\x12@'
b'\x06\x00\x00\x00333333\x1f@\x00\x00\x00\x00\x00\x00"@'
b'\x0c\x00\x00\x00\xcd\xcc\xcc\xcc\xcc*\x9a\x99\x99\x99YL@'
>>>

```

如你所见，创建一个可迭代对象的一个原因是它能允许使用一个生成器推导来创建记录。如果你不使用这种技术，那么代码可能会像下面这样：


```
def read_records(format, f):
    record_struct = Struct(format)
    while True:
        chk = f.read(record_struct.size)
        if chk == b'':
            break
        yield record_struct.unpack(chk)
```

在函数 `unpack_records()` 中使用了另外一种方法 `unpack_from()`。`unpack_from()` 对于从一个大型二进制数组中提取二进制数据非常有用，因为它不会产生任何的临时对象或者进行内存复制操作。你只需要给它一个字节字符串(或数组)和一个字节偏移量，它会从那个位置开始直接解包数据。

如果你使用 `unpack()` 来代替 `unpack_from()`，你需要修改代码来构造大量的小的切片以及进行偏移量的计算。比如：

```
def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack(data[offset:offset + record_struct.size])
            for offset in range(0, len(data), record_struct.size))
```

这种方案除了代码看上去很复杂外，还得做很多额外的工作，因为它执行了大量的偏移量计算，复制数据以及构造小的切片对象。如果你准备从读取到的一个大型字节字符串中解包大量的结构体的话，`unpack_from()` 会表现的更出色。

在解包的时候，`collections` 模块中的命名元组对象或许是你想要用到的。它可以让你给返回元组设置属性名称。例如：

```
from collections import namedtuple

Record = namedtuple('Record', ['kind', 'x', 'y'])

with open('data.p', 'rb') as f:
    records = (Record(*r) for r in read_records('<idd', f))

for r in records:
    print(r.kind, r.x, r.y)
```

如果你的程序需要处理大量的二进制数据，你最好使用 `numpy` 模块。例如，你可以将一个二进制数据读取到一个结构化数组中而不是一个元组列表中。就像下面这样：

```

>>> import numpy as np
>>> f = open('data.b', 'rb')
>>> records = np.fromfile(f, dtype='<i,<d,<d')
>>> records
array([(1, 2.3, 4.5), (6, 7.8, 9.0), (12, 13.4, 56.7)],
      dtype=[('f0', '<i4'), ('f1', '<f8'), ('f2', '<f8')])
>>> records[0]
(1, 2.3, 4.5)
>>> records[1]
(6, 7.8, 9.0)
>>>

```

最后提一点，如果你需要从已知的文件格式(如图片格式，图形文件，HDF5等)中读取二进制数据时，先检查看看Python是不是已经提供了现存的模块。因为不到万不得已没有必要去重复造轮子。

6.12 读取嵌套和可变长二进制数据

问题

你需要读取包含嵌套或者可变长记录集合的复杂二进制格式的数据。这些数据可能包含图片、视频、电子地图文件等。

解决方案

`struct` 模块可被用来编码/解码几乎所有类型的二进制的数据结构。为了解释清楚这种数据，假设你用下面的Python数据结构来表示一个组成一系列多边形的点的集合：

现在假设这个数据被编码到一个以下列头部开始的二进制文件中去了：

```

+-----+-----+-----+-----+
|Byte  | Type  | Description          |
+=====+=====+=====+=====+
|0     | int   | File code (0x1234,  |
+-----+-----+-----+-----+
|4     | double | Minimum x (little  |
+-----+-----+-----+-----+
|12    | double | Minimum y (little  |
+-----+-----+-----+-----+
|20    | double | Maximum x (little  |
+-----+-----+-----+-----+
|28    | double | Maximum y (little  |
+-----+-----+-----+-----+
|36    | int   | Number of polygons |
+-----+-----+-----+-----+

```

紧跟着头部是一系列的多边形记录，编码格式如下：

```
+-----+-----+-----+
|Byte  | Type  | Description                               |
+=====+=====+=====+
|0     | int   | Record length including length (N bytes) |
+-----+-----+-----+
|4-N   | Points| Pairs of (X,Y) coords as doubles        |
+-----+-----+-----+
```

为了写这样的文件，你可以使用如下的Python代码：

```
import struct
import itertools

def write_polys(filename, polys):
    # Determine bounding box
    flattened = list(itertools.chain(*polys))
    min_x = min(x for x, y in flattened)
    max_x = max(x for x, y in flattened)
    min_y = min(y for x, y in flattened)
    max_y = max(y for x, y in flattened)
    with open(filename, 'wb') as f:
        f.write(struct.pack('<iddddi', 0x1234,
                           min_x, min_y,
                           max_x, max_y,
                           len(polys)))
    for poly in polys:
        size = len(poly) * struct.calcsize('<dd')
        f.write(struct.pack('<i', size + 4))
        for pt in poly:
            f.write(struct.pack('<dd', *pt))
```

将数据读取回来的时候，可以利用函数 `struct.unpack()`，代码很相似，基本就是上面写操作的逆序。如下：

```

def read_polys(filename):
    with open(filename, 'rb') as f:
        # Read the header
        header = f.read(40)
        file_code, min_x, min_y, max_x, max_y, num_polys = \
            struct.unpack('<iiddi', header)
        polys = []
        for n in range(num_polys):
            pbytes, = struct.unpack('<i', f.read(4))
            poly = []
            for m in range(pbytes // 16):
                pt = struct.unpack('<dd', f.read(16))
                poly.append(pt)
            polys.append(poly)
    return polys

```

尽管这个代码可以工作，但是里面混杂了很多读取、解包数据结构和其他细节的代码。如果用这样的代码来处理真实的数据文件，那未免也太繁杂了点。因此很显然应该有另一种解决方法可以简化这些步骤，让程序员只关注自最重要的事情。

在本小节接下来的部分，我会逐步演示一个更加优秀的解析字节数据的方案。目标是给程序员提供一个高级的文件格式化方法，并简化读取和解包数据的细节。但是我要先提醒你，本小节接下来的部分代码应该是整本书中最复杂最高级的例子，使用了大量的面向对象编程和元编程技术。一定要仔细阅读我们的讨论部分，另外也要参考下其他章节内容。

首先，当读取字节数据的时候，通常在文件开始部分会包含文件头和其他的数据结构。尽管struct模块可以解包这些数据到一个元组中去，另外一种表示这种信息的方式就是使用一个类。就像下面这样：

```

import struct

class StructField:
    """
    Descriptor representing a simple structure field
    """
    def __init__(self, format, offset):
        self.format = format
        self.offset = offset
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            r = struct.unpack_from(self.format, instance._buffer, self.offset)
            return r[0] if len(r) == 1 else r

class Structure:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)

```

这里我们使用了一个描述器来表示每个结构字段，每个描述器包含一个结构兼容格式的代码以及一个字节偏移量，存储在内部的内存缓冲中。在 `__get__()` 方法中，`struct.unpack_from()` 函数被用来从缓冲中解包一个值，省去了额外的分片或复制操作步骤。

`Structure` 类就是一个基础类，接受字节数据并存储在内部的内存缓冲中，并被 `StructField` 描述器使用。这里使用了 `memoryview()`，我们会在后面详细讲解它是用来干嘛的。

使用这个代码，你现在就能定义一个高层次的结构对象来表示上面表格信息所期望的文件格式。例如：

```
class PolyHeader(Structure):
    file_code = StructField('<i', 0)
    min_x = StructField('<d', 4)
    min_y = StructField('<d', 12)
    max_x = StructField('<d', 20)
    max_y = StructField('<d', 28)
    num_polys = StructField('<i', 36)
```

下面的例子利用这个类来读取之前我们写入的多边形数据的头部数据：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader(f.read(40))
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

这个很有趣，不过这种方式还是有一些烦人的地方。首先，尽管你获得了一个类接口的便利，但是这个代码还是有点臃肿，还需要使用者指定很多底层的细节(比如重复使用 `StructField`，指定偏移量等)。另外，返回的结果类同样确实一些便利的方法来计算结构的总数。

任何时候只要你遇到了像这样冗余的类定义，你应该考虑下使用类装饰器或元类。元类有一个特性就是它能够被用来填充许多低层的实现细节，从而释放使用者的负担。下面我来举个例子，使用元类稍微改造下我们的 `Structure` 类：

```
class StructureMeta(type):
    ...
    Metaclass that automatically creates StructField descriptors
    ...
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if format.startswith(('<', '>', '!', '@')):
                byte_order = format[0]
                format = format[1:]
            format = byte_order + format
            setattr(self, fieldname, StructField(format, offset))
            offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)

class Structure(metaclass=StructureMeta):
    def __init__(self, bytedata):
        self._buffer = bytedata

    @classmethod
    def from_file(cls, f):
        return cls(f.read(cls.struct_size))
```

使用新的 `Structure` 类，你可以像下面这样定义一个结构：

```
class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        ('d', 'min_x'),
        ('d', 'min_y'),
        ('d', 'max_x'),
        ('d', 'max_y'),
        ('i', 'num_polys')
    ]
```

正如你所见，这样写就简单多了。我们添加的类方法 `from_file()` 让我们在不需知道任何数据的大小和结构的情况下就能轻松的从文件中读取数据。比如：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

一旦你开始使用了元类，你就可以让它变得更加智能。例如，假设你还想支持嵌套的字节结构，下面是对前面元类的一个小的改进，提供了一个新的辅助描述器来达到想要的效果：

```

class NestedStruct:
    """
    Descriptor representing a nested structure
    """
    def __init__(self, name, struct_type, offset):
        self.name = name
        self.struct_type = struct_type
        self.offset = offset

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            data = instance._buffer[self.offset:
                                     self.offset+self.struct_type.struct_size]
            result = self.struct_type(data)
            # Save resulting structure back on instance to avoid
            # further recomputation of this step
            setattr(instance, self.name, result)
            return result

class StructureMeta(type):
    """
    Metaclass that automatically creates StructField descriptors
    """
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if isinstance(format, StructureMeta):
                setattr(self, fieldname,
                        NestedStruct(fieldname, format, offset))
                offset += format.struct_size
            else:
                if format.startswith('<', '>', '!', '@'):
                    byte_order = format[0]
                    format = format[1:]
                format = byte_order + format
                setattr(self, fieldname, StructField(format, offset))
                offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)

```

在这段代码中，`NestedStruct` 描述器被用来叠加另外一个定义在某个内存区域上的结构。它通过将原始内存缓冲进行切片操作后实例化给定的结构类型。由于底层的内存缓冲区是通过一个内存视图初始化的，所以这种切片操作不会引发任何的额外的内存复制。相反，它仅仅就是之前的内存的一个叠加而已。另外，为了防止重复实例化，通过使用和8.10小节同样的技术，描述器保存了该实例中的内部结构对象。

使用这个新的修正版，你就可以像下面这样编写：


```

class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'), # nested struct
        (Point, 'max'), # nested struct
        ('i', 'num_polys')
    ]

```

令人惊讶的是，它也能按照预期的正常工作，我们实际操作下：

```

>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min # Nested structure
<__main__.Point object at 0x1006a48d0>
>>> phead.min.x
0.5
>>> phead.min.y
0.5
>>> phead.max.x
7.0
>>> phead.max.y
9.2
>>> phead.num_polys
3
>>>

```

到目前为止，一个处理定长记录的框架已经写好了。但是如果组件记录是变长的呢？比如，多边形文件包含变长的部分。

一种方案是写一个类来表示字节数据，同时写一个工具函数来通过多少方式解析内容。跟6.11小节的代码很类似：

```

class SizedRecord:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)

    @classmethod
    def from_file(cls, f, size_fmt, includes_size=True):
        sz_nbytes = struct.calcsize(size_fmt)
        sz_bytes = f.read(sz_nbytes)
        sz, = struct.unpack(size_fmt, sz_bytes)
        buf = f.read(sz - includes_size * sz_nbytes)
        return cls(buf)

    def iter_as(self, code):
        if isinstance(code, str):
            s = struct.Struct(code)
            for off in range(0, len(self._buffer), s.size):
                yield s.unpack_from(self._buffer, off)
        elif isinstance(code, StructureMeta):
            size = code.struct_size
            for off in range(0, len(self._buffer), size):
                data = self._buffer[off:off+size]
                yield code(data)

```

类方法 `SizedRecord.from_file()` 是一个工具，用来从一个文件中读取带大小前缀的数据块，这也是很多文件格式常用的方式。作为输入，它接受一个包含大小编码的结构格式编码，并且也是自己形式。可选的 `includes_size` 参数指定了字节数是否包含头部大小。下面是一个例子教你怎样使用从多边形文件中读取单独的多边形数据：

```

>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.num_polys
3
>>> polydata = [ SizedRecord.from_file(f, '<i')
...               for n in range(phead.num_polys) ]
>>> polydata
[<__main__.SizedRecord object at 0x1006a4d50>,
<__main__.SizedRecord object at 0x1006a4f50>,
<__main__.SizedRecord object at 0x10070da90>]
>>>

```

可以看出，`SizedRecord` 实例的内容还没有被解析出来。可以使用 `iter_as()` 方法来达到目的，这个方法接受一个结构格式化编码或者是 `Structure` 类作为输入。这样子可以很灵活的去解析数据，例如：

```

>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as('<dd'):
...         print(p)
...
Polygon 0
(1.0, 2.5)
(3.5, 4.0)
(2.5, 1.5)
Polygon 1
(7.0, 1.2)
(5.1, 3.0)
(0.5, 7.5)
(0.8, 9.0)
Polygon 2
(3.4, 6.3)
(1.2, 0.5)
(4.6, 9.2)
>>>

```

```

>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as(Point):
...         print(p.x, p.y)
...
Polygon 0
1.0 2.5
3.5 4.0
2.5 1.5
Polygon 1
7.0 1.2
5.1 3.0
0.5 7.5
0.8 9.0
Polygon 2
3.4 6.3
1.2 0.5
4.6 9.2
>>>

```

将所有这些结合起来，下面是一个 `read_polys()` 函数的另外一个修正版：

```

class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'),
        (Point, 'max'),
        ('i', 'num_polys')
    ]

def read_polys(filename):
    polys = []
    with open(filename, 'rb') as f:
        phead = PolyHeader.from_file(f)
        for n in range(phead.num_polys):
            rec = SizedRecord.from_file(f, '<i')
            poly = [ (p.x, p.y) for p in rec.iter_as(Point) ]
            polys.append(poly)
    return polys

```

讨论

这一节向你展示了许多高级的编程技术，包括描述器，延迟计算，元类，类变量和内存视图。然而，它们都为了同一个特定的目标服务。

上面的实现的一个主要特征是它是基于懒解包的思想。当一个 `Structure` 实例被创建时，`__init__()` 仅仅只是创建一个字节数据的内存视图，没有做其他任何事。特别的，这时候并没有任何的解包或者其他与结构相关的操作发生。这样做的一个动机是你可能仅仅只对一个字节记录的某一小部分感兴趣。我们只需要解包你需要访问的部分，而不是整个文件。

为了实现懒解包和打包，需要使用 `StructField` 描述器类。用户在 `_fields_` 中列出来的每个属性都会被转化成一个 `StructField` 描述器，它将相关结构格式码和偏移值保存到存储缓存中。元类 `StructureMeta` 在多个结构类被定义时自动创建了这些描述器。我们使用元类的一个主要原因是它使得用户非常方便的通过一个高层描述就能指定结构格式，而无需考虑低层的细节问题。

`StructureMeta` 的一个很微妙的地方就是它会固定字节数据顺序。也就是说，如果任意的属性指定了一个字节顺序(<表示低位优先 或者 >表示高位优先)，那后面所有字段的顺序都以这个顺序为准。这么做可以帮助避免额外输入，但是在定义的中间我们仍然可能切换顺序的。比如，你可能有一些比较复杂的结构，就像下面这样：

```
class ShapeFile(Structure):
    _fields_ = [ ('>i', 'file_code'), # Big endian
                ('20s', 'unused'),
                ('i', 'file_length'),
                ('<i', 'version'), # Little endian
                ('i', 'shape_type'),
                ('d', 'min_x'),
                ('d', 'min_y'),
                ('d', 'max_x'),
                ('d', 'max_y'),
                ('d', 'min_z'),
                ('d', 'max_z'),
                ('d', 'min_m'),
                ('d', 'max_m') ]
```

之前我们提到过，`memoryview()` 的使用可以帮助我们避免内存的复制。当结构存在嵌套的时候，`memoryviews` 可以叠加同一内存区域上定义的机构的不同部分。这个特性比较微妙，但是它关注的是内存视图与普通字节数组的切片操作行为。如果你在一个字节字符串或字节数组上执行切片操作，你通常会得到一个数据的拷贝。而内存视图切片不是这样的，它仅仅是在已存在的内存上面叠加而已。因此，这种方式更加高效。

还有很多相关的章节可以帮助我们扩展这里讨论的方案。参考8.13小节使用描述器构建一个类型系统。8.10小节有更多关于延迟计算属性值的讨论，并且跟NestredStruct描述器的实现也有关。9.19小节有一个使用元类来初始化类成员的例子，和 `StructureMeta` 类非常相似。Python的 `ctypes` 源码同样也很有趣，它提供了对定义数据结构、数据结构嵌套这些相似功能的支持。

6.13 数据的累加与统计操作

问题

你需要处理一个很大的数据集并需要计算数据总和或其他统计量。

解决方案

对于任何涉及到统计、时间序列以及其他相关技术的数据分析问题，都可以考虑使用 [Pandas库](#)。

为了让你先体验下，下面是一个使用Pandas来分析芝加哥城市的 [老鼠和啮齿类动物数据库](#) 的例子。在我写这篇文章的时候，这个数据库是一个拥有大概74,000行数据的CSV文件。

```

>>> import pandas

>>> # Read a CSV file, skipping last line
>>> rats = pandas.read_csv('rats.csv', skip_footer=1)
>>> rats
<class 'pandas.core.frame.DataFrame'>
Int64Index: 74055 entries, 0 to 74054
Data columns:
Creation Date 74055 non-null values
Status 74055 non-null values
Completion Date 72154 non-null values
Service Request Number 74055 non-null values
Type of Service Request 74055 non-null values
Number of Premises Baited 65804 non-null values
Number of Premises with Garbage 65600 non-null values
Number of Premises with Rats 65752 non-null values
Current Activity 66041 non-null values
Most Recent Action 66023 non-null values
Street Address 74055 non-null values
ZIP Code 73584 non-null values
X Coordinate 74043 non-null values
Y Coordinate 74043 non-null values
Ward 74044 non-null values
Police District 74044 non-null values
Community Area 74044 non-null values
Latitude 74043 non-null values
Longitude 74043 non-null values
Location 74043 non-null values
dtypes: float64(11), object(9)

>>> # Investigate range of values for a certain field
>>> rats['Current Activity'].unique()
array([nan, Dispatch Crew, Request Sanitation Inspector], dtype=object)
>>> # Filter the data
>>> crew_dispatched = rats[rats['Current Activity'] == 'Dispatch Crew']
>>> len(crew_dispatched)
65676
>>>

>>> # Find 10 most rat-infested ZIP codes in Chicago
>>> crew_dispatched['ZIP Code'].value_counts()[:10]
60647 3837
60618 3530
60614 3284
60629 3251
60636 2801
60657 2465
60641 2238
60609 2206
60651 2152
60632 2071
>>>

>>> # Group by completion date
>>> dates = crew_dispatched.groupby('Completion Date')
<pandas.core.groupby.DataFrameGroupBy object at 0x10d0a2a10>
>>> len(dates)

```

```
472
>>>

>>> # Determine counts on each day
>>> date_counts = dates.size()
>>> date_counts[0:10]
Completion Date
01/03/2011 4
01/03/2012 125
01/04/2011 54
01/04/2012 38
01/05/2011 78
01/05/2012 100
01/06/2011 100
01/06/2012 58
01/07/2011 1
01/09/2012 12
>>>

>>> # Sort the counts
>>> date_counts.sort()
>>> date_counts[-10:]
Completion Date
10/12/2012 313
10/21/2011 314
09/20/2011 316
10/26/2011 319
02/22/2011 325
10/26/2012 333
03/17/2011 336
10/13/2011 378
10/14/2011 391
10/07/2011 457
>>>
```

嗯，看样子2011年10月7日对老鼠们来说是个很忙碌的日子啊！^_^

讨论

Pandas是一个拥有很多特性的大型函数库，我在这里不可能介绍完。但是只要你需要去分析大型数据集、对数据分组、计算各种统计量或其他类似任务的话，这个函数库真的值得你去看一看。

第七章：函数

使用 `def` 语句定义函数是所有程序的基础。本章的目标是讲解一些更加高级和不常见的函数定义与使用模式。涉及到的内容包括默认参数、任意数量参数、强制关键字参数、注解和闭包。另外，一些高级的控制流和利用回调函数传递数据的技术在这里也会讲解

到。

Contents:

7.1 可接受任意数量参数的函数

问题

你想构造一个可接受任意数量参数的函数。

解决方案

为了能让一个函数接受任意数量的位置参数，可以使用一个*参数。例如：

```
def avg(first, *rest):
    return (first + sum(rest)) / (1 + len(rest))

# Sample use
avg(1, 2) # 1.5
avg(1, 2, 3, 4) # 2.5
```

在这个例子中，`rest`是由所有其他位置参数组成的元组。然后我们在代码中把它当成了一个序列来进行后续的计算。

为了接受任意数量的关键字参数，使用一个以**开头的参数。比如：

```
import html

def make_element(name, value, **attrs):
    keyvals = [' %s="%s"' % item for item in attrs.items()]
    attr_str = ''.join(keyvals)
    element = '<{name}{attrs}>{value}</{name}>'.format(
        name=name,
        attrs=attr_str,
        value=html.escape(value))
    return element

# Example
# Creates '<item size="large" quantity="6">Albatross</item>'
make_element('item', 'Albatross', size='large', quantity=6)

# Creates '<p>&lt;spam&gt;</p>'
make_element('p', '<spam>')
```


在这里，`attrs`是一个包含所有被传入进来的关键字参数的字典。

如果你还希望某个函数能同时接受任意数量的位置参数和关键字参数，可以同时使用`*`和`**`。比如：

```
def anyargs(*args, **kwargs):  
    print(args) # A tuple  
    print(kwargs) # A dict
```

使用这个函数时，所有位置参数会被放到`args`元组中，所有关键字参数会被放到字典`kwargs`中。

讨论

一个`*`参数只能出现在函数定义中最后一个位置参数后面，而`**`参数只能出现在最后一个参数。有一点要注意的是，在`*`参数后面仍然可以定义其他参数。

```
def a(x, *args, y):  
    pass  
  
def b(x, *args, y, **kwargs):  
    pass
```

这种参数就是我们所说的强制关键字参数，在后面7.2小节还会详细讲解到。

7.2 只接受关键字参数的函数

问题

你希望函数的某些参数强制使用关键字参数传递

解决方案

将强制关键字参数放到某个`*`参数或者当个`*`后面就能达到这种效果。比如：

```
def recv(maxsize, *, block):
    'Receives a message'
    pass

recv(1024, True) # TypeError
recv(1024, block=True) # Ok
```

利用这种技术，我们还能在接受任意多个位置参数的函数中指定关键字参数。比如：

```
def minimum(*values, clip=None):
    m = min(values)
    if clip is not None:
        m = clip if clip > m else m
    return m

minimum(1, 5, 2, -5, 10) # Returns -5
minimum(1, 5, 2, -5, 10, clip=0) # Returns 0
```

讨论

很多情况下，使用强制关键字参数会比使用位置参数表意更加清晰，程序也更加具有可读性。例如，考虑下如下一个函数调用：

```
msg = recv(1024, False)
```

如果调用者对`recv`函数并不是很熟悉，那他肯定不明白那个`False`参数到底来干嘛用的。但是，如果代码变成下面这样子的话就清楚多了：

```
msg = recv(1024, block=False)
```

另外，使用强制关键字参数也会比使用`**kwargs`参数更好，因为在使用函数`help`的时候输出也会更容易理解：

```
>>> help(recv)
Help on function recv in module __main__:
recv(maxsize, *, block)
    Receives a message
```

强制关键字参数在一些更高级场合同样也很有用。例如，它们可以被用来在使用*args和**kwargs参数作为输入的函数中插入参数，9.11小节有一个这样的例子。

7.3 给函数参数增加元信息

问题

你写好了一个函数，然后想为这个函数的参数增加一些额外的信息，这样的话其他使用者就能清楚的知道这个函数应该怎么使用。

解决方案

使用函数参数注解是一个很好的办法，它能提示程序员应该怎样正确使用这个函数。例如，下面有一个被注解了的函数：

```
def add(x:int, y:int) -> int:
    return x + y
```

python解释器不会对这些注解添加任何的语义。它们不会被类型检查，运行时跟没有加注解之前的效果也没有任何差距。然而，对于那些阅读源码的人来讲就很有帮助啦。第三方工具和框架可能会对这些注解添加语义。同时它们也会出现在文档中。

```
>>> help(add)
Help on function add in module __main__:
add(x: int, y: int) -> int
>>>
```

尽管你可以使用任意类型的对象给函数添加注解(例如数字，字符串，对象实例等等)，不过通常来讲使用类或者字符串会比较好点。

讨论

函数注解只存储在函数的 `__annotations__` 属性中。例如：

```
>>> add.__annotations__
{'y': <class 'int'>, 'return': <class 'int'>, 'x': <class 'int'>}
```

尽管注解的使用方法可能有很多种，但是它们的主要用途还是文档。因为python并没有类型声明，通常来讲仅仅通过阅读源码很难知道应该传递什么样的参数给这个函数。这时候使用注解就能给程序员更多的提示，让他们可以争取的使用函数。

参考9.20小节的一个更加高级的例子，演示了如何利用注解来实现多分派(比如重载函数)。

7.4 返回多个值的函数

问题

你希望构造一个可以返回多个值的函数

解决方案

为了能返回多个值，函数直接return一个元组就行了。例如：

```
>>> def myfun():
...     return 1, 2, 3
...
>>> a, b, c = myfun()
>>> a
1
>>> b
2
>>> c
3
```

讨论

尽管myfun()看上去返回了多个值，实际上是先创建了一个元组然后返回的。这个语法看上去比较奇怪，实际上我们使用的是逗号来生成一个元组，而不是用括号。比如下面的：

```
>>> a = (1, 2) # With parentheses
>>> a
(1, 2)
>>> b = 1, 2 # Without parentheses
>>> b
(1, 2)
>>>
```

当我们调用返回一个元组的函数的时候，通常我们会将结果赋值给多个变量，就像上面的那样。其实这就是1.1小节中我们所说的元组解包。返回结果也可以赋值给单个变量，这时候这个变量值就是函数返回的那个元组本身了：

```
>>> x = myfun()
>>> x
(1, 2, 3)
>>>
```

7.5 定义有默认参数的函数

问题

你想定义一个函数或者方法，它的一个或多个参数是可选的并且有一个默认值。

解决方案

定义一个有可选参数的函数是非常简单的，直接在函数定义中给参数指定一个默认值，并放到参数列表最后就行了。例如：

```
def spam(a, b=42):
    print(a, b)

spam(1) # Ok. a=1, b=42
spam(1, 2) # Ok. a=1, b=2
```

如果默认参数是一个可修改的容器比如一个列表、集合或者字典，可以使用None作为默认值，就像下面这样：

```
# Using a list as a default value
def spam(a, b=None):
    if b is None:
        b = []
    ...
```

如果你并不想提供一个默认值，而是想仅仅测试下某个默认参数是不是有传递进来，可以像下面这样写：

```
_no_value = object()

def spam(a, b=_no_value):
    if b is _no_value:
        print('No b value supplied')
    ...
```

我们测试下这个函数：

```
>>> spam(1)
No b value supplied
>>> spam(1, 2) # b = 2
>>> spam(1, None) # b = None
>>>
```

仔细观察可以发现到传递一个None值和不传值两种情况是有差别的。

讨论

定义带默认值参数的函数是很简单的，但绝不仅仅只是这个，还有一些东西在这里也深入讨论下。

首先，默认参数的值仅仅在函数定义的时候赋值一次。试着运行下面这个例子：

```
>>> x = 42
>>> def spam(a, b=x):
...     print(a, b)
...
>>> spam(1)
1 42
>>> x = 23 # Has no effect
>>> spam(1)
1 42
>>>
```

注意到当我们改变x的值的时候对默认参数值并没有影响，这是因为在函数定义的时候就已经确定了它的默认值了。

其次，默认参数的值应该是不可变的对象，比如None、True、False、数字或字符串。特别的，千万不要像下面这样写代码：

```
def spam(a, b=[]): # NO!  
    ...
```

如果你这么做了，当默认值在其他地方被修改后你将会遇到各种麻烦。这些修改会影响到下次调用这个函数时的默认值。比如：

```
>>> def spam(a, b=[]):  
...     print(b)  
...     return b  
...  
>>> x = spam(1)  
>>> x  
[]  
>>> x.append(99)  
>>> x.append('Yow!')  
>>> x  
[99, 'Yow!']  
>>> spam(1) # Modified List gets returned!  
[99, 'Yow!']  
>>>
```

这种结果应该不是你想要的。为了避免这种情况的发生，最好是将默认值设为None，然后在函数里面检查它，前面的例子就是这样做的。

在测试None值时使用 `is` 操作符是很重要的，也是这种方案的关键点。有时候大家会犯下下面这样的错误：

```
def spam(a, b=None):  
    if not b: # NO! Use 'b is None' instead  
        b = []  
    ...
```

这么写的问题在于尽管None值确实是被当成False，但是还有其他的对象(比如长度为0的字符串、列表、元组、字典等)都会被当做False。因此，上面的代码会误将一些其他输入也当成是没有输入。比如：

```
>>> spam(1) # OK  
>>> x = []  
>>> spam(1, x) # Silent error. x value overwritten by default  
>>> spam(1, 0) # Silent error. 0 ignored  
>>> spam(1, '') # Silent error. '' ignored  
>>>
```

最后一个问题比较微妙，那就是一个函数需要测试某个可选参数是否被使用者传递进来。这时候需要小心的是你不能用某个默认值比如None、0或者False值来测试用户提供的值(因为这些值都是合法的值，是可能被用户传递进来的)。因此，你需要其他的解决方案了。

为了解决这个问题，你可以创建一个独一无二的私有对象实例，就像上面的_no_value变量那样。在函数里面，你可以通过检查被传递参数值跟这个实例是否一样来判断。这里的思路是用户不可能去传递这个_no_value实例作为输入。因此，这里通过检查这个值就能确定某个参数是否被传递进来了。

这里对object()的使用看上去有点不太常见。object是python中所有类的基类。你可以创建object类的实例，但是这些实例没什么实际用处，因为它没有任何有用的方法，也没有任何实例数据(因为它没有任何的实例字典，你甚至都不能设置任何属性值)。你唯一能做的就是测试同一性。这个刚好符合我的要求，因为我在函数中就只需要一个同一性的测试而已。

7.6 定义匿名或内联函数

问题

你想为sort()操作创建一个很短的回调函数，但又不想用def去写一个单行函数，而是希望通过某个快捷方式以内联方式来创建这个函数。

解决方案

当一些函数很简单，仅仅只是计算一个表达式的值的时候，就可以使用lambda表达式来代替了。比如：

```
>>> add = lambda x, y: x + y
>>> add(2,3)
5
>>> add('hello', 'world')
'helloworld'
>>>
```

这里使用的lambda表达式跟下面的效果是一样的：


```
>>> def add(x, y):
...     return x + y
...
>>> add(2,3)
5
>>>
```

lambda表达式典型的使用场景是排序或数据reduce等:

```
>>> names = ['David Beazley', 'Brian Jones',
...          'Raymond Hettinger', 'Ned Batchelder']
>>> sorted(names, key=lambda name: name.split()[-1].lower())
['Ned Batchelder', 'David Beazley', 'Raymond Hettinger', 'Brian Jones']
>>>
```

讨论

尽管lambda表达式允许你定义简单函数，但是它的使用是有限制的。你只能指定单个表达式，它的值就是最后的返回值。也就是说不能包含其他的语言特性了，包括多个语句、条件表达式、迭代以及异常处理等等。

你可以不使用lambda表达式就能编写大部分python代码。但是，当有人编写大量计算表达式值的短小函数或者需要用户提供回调函数的程序的时候，你就会看到lambda表达式的身影了。

7.7 匿名函数捕获变量值

问题

你用lambda定义了一个匿名函数，并想在定义时捕获到某些变量的值。

解决方案

先看下下面代码的效果:

```
>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>>
```

现在我问你，`a(10)`和`b(10)`返回的结果是什么？如果你认为结果是20和30，那么你就错了：

```
>>> a(10)
30
>>> b(10)
30
>>>
```

这其中的奥妙在于`lambda`表达式中的`x`是一个自由变量，在运行时绑定值，而不是定义时就绑定，这跟函数的默认值参数定义是不同的。因此，在调用这个`lambda`表达式的时候，`x`的值是执行时的值。例如：

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

如果你想让某个匿名函数在定义时就捕获到值，可以将那个参数值定义成默认参数即可，就像下面这样：

```
>>> x = 10
>>> a = lambda y, x=x: x + y
>>> x = 20
>>> b = lambda y, x=x: x + y
>>> a(10)
20
>>> b(10)
30
>>>
```

讨论

在这里列出来的问题是新手很容易犯的错误，有些新手可能会不恰当的lambda表达式。比如，通过在一个循环或列表推导中创建一个lambda表达式列表，并期望函数能在定义时就记住每次的迭代值。例如：

```
>>> funcs = [lambda x: x+n for n in range(5)]
>>> for f in funcs:
...     print(f(0))
...
4
4
4
4
4
>>>
```

但是实际效果是运行是n的值为迭代的最后一个值。现在我们用另一种方式修改一下：

```
>>> funcs = [lambda x, n=n: x+n for n in range(5)]
>>> for f in funcs:
...     print(f(0))
...
0
1
2
3
4
>>>
```

通过使用函数默认值参数形式，lambda函数在定义时就能绑定到值。

7.8 减少可调用对象的参数个数

问题

你有一个被其他python代码使用的callable对象，可能是一个回调函数或者是一个处理器，但是它的参数太多了，导致调用时出错。

解决方案

如果需要减少某个函数的参数个数，你可以使用 `functools.partial()`。 `partial()` 函数允许你给一个或多个参数设置固定的值，减少接下来被调用时的参数个数。为了演示清楚，假设你有下面这样的函数：

```
def spam(a, b, c, d):  
    print(a, b, c, d)
```

现在我们使用 `partial()` 函数来固定某些参数值：

```
>>> from functools import partial  
>>> s1 = partial(spam, 1) # a = 1  
>>> s1(2, 3, 4)  
1 2 3 4  
>>> s1(4, 5, 6)  
1 4 5 6  
>>> s2 = partial(spam, d=42) # d = 42  
>>> s2(1, 2, 3)  
1 2 3 42  
>>> s2(4, 5, 5)  
4 5 5 42  
>>> s3 = partial(spam, 1, 2, d=42) # a = 1, b = 2, d = 42  
>>> s3(3)  
1 2 3 42  
>>> s3(4)  
1 2 4 42  
>>> s3(5)  
1 2 5 42  
>>>
```

可以看出 `partial()` 固定某些参数并返回一个新的callable对象。这个新的callable接受未赋值的参数，然后跟之前已经赋值过的参数合并起来，最后将所有参数传递给原始函数。

讨论

本节要解决的问题是让原本不兼容的代码可以一起工作。下面我会列举一系列的例子。

第一个例子是，假设你有一个点的列表来表示(x,y)坐标元组。你可以使用下面的函数来计算两点之间的距离：

```
points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]
```

```
import math
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.hypot(x2 - x1, y2 - y1)
```

现在假设你想以某个点为基点，根据点和基点之间的距离来排序所有的这些点。列表的 `sort()` 方法接受一个关键字参数来自定义排序逻辑，但是它只能接受一个单个参数的函数(`distance()`很明显是不符合条件的)。现在我们可以通过使用 `partial()` 来解决这个问题：

```
>>> pt = (4, 3)
>>> points.sort(key=partial(distance,pt))
>>> points
[(3, 4), (1, 2), (5, 6), (7, 8)]
>>>
```

更进一步，`partial()` 通常被用来微调其他库函数所使用的回调函数的参数。例如，下面是一段代码，使用 `multiprocessing` 来异步计算一个结果值，然后这个值被传递给一个接受一个`result`值和一个可选`logging`参数的回调函数：

```
def output_result(result, log=None):
    if log is not None:
        log.debug('Got: %r', result)

# A sample function
def add(x, y):
    return x + y

if __name__ == '__main__':
    import logging
    from multiprocessing import Pool
    from functools import partial

    logging.basicConfig(level=logging.DEBUG)
    log = logging.getLogger('test')

    p = Pool()
    p.apply_async(add, (3, 4), callback=partial(output_result, log=log))
    p.close()
    p.join()
```

当给 `apply_async()` 提供回调函数时，通过使用 `partial()` 传递额外的 `logging` 参数。而 `multiprocessing` 对这些一无所知——它仅仅只是使用单个值来调用回调函数。

作为一个类似的例子，考虑下编写网络服务器的问题，`socketserver` 模块让它变得很容易。下面是个简单的echo服务器：

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        for line in self.rfile:
            self.wfile.write(b'GOT:' + line)

serv = TCPServer(('', 15000), EchoHandler)
serv.serve_forever()
```

不过，假设你想给EchoHandler增加一个可以接受其他配置选项的 `__init__` 方法。比如：

```
class EchoHandler(StreamRequestHandler):
    # ack is added keyword-only argument. *args, **kwargs are
    # any normal parameters supplied (which are passed on)
    def __init__(self, *args, ack, **kwargs):
        self.ack = ack
        super().__init__(*args, **kwargs)

    def handle(self):
        for line in self.rfile:
            self.wfile.write(self.ack + line)
```

这么修改后，我们就不需要显式地在TCPServer类中添加前缀了。但是你再次运行程序后会报类似下面的错误：

```
Exception happened during processing of request from ('127.0.0.1', 59834)
Traceback (most recent call last):
...
TypeError: __init__() missing 1 required keyword-only argument: 'ack'
```

初看起来好像很难修正这个错误，除了修改 `socketserver` 模块源代码或者使用某些奇怪的方法之外。但是，如果使用 `partial()` 就能很轻松的解决——给它传递 `ack` 参数的值来初始化即可，如下：

```
from functools import partial
serv = TCPServer('', 15000), partial(EchoHandler, ack=b'RECEIVED:')
serv.serve_forever()
```

在这个例子中，`__init__()` 方法中的`ack`参数声明方式看上去很有趣，其实就是声明`ack`为一个强制关键字参数。关于强制关键字参数问题我们在7.2小节我们已经讨论过了，读者可以再去回顾一下。

很多时候 `partial()` 能实现的效果，`lambda`表达式也能实现。比如，之前的几个例子可以使用下面这样的表达式：

```
points.sort(key=lambda p: distance(pt, p))
p.apply_async(add, (3, 4), callback=lambda result: output_result(result, log))
serv = TCPServer('', 15000),
        lambda *args, **kwargs: EchoHandler(*args, ack=b'RECEIVED:', **kwargs))
```

这样写也能实现同样的效果，不过相比而已会显得比较臃肿，对于阅读代码的人来讲也更加难懂。这时候使用 `partial()` 可以更加直观的表达你的意图(给某些参数预先赋值)。

7.9 将单方法的类转换为函数

问题

你有一个除 `__init__()` 方法外只定义了一个方法的类。为了简化代码，你想将它转换成一个函数。

解决方案

大多数情况下，可以使用闭包来将单个方法的类转换成函数。举个例子，下面示例中的类允许使用者根据某个模板方案来获取到URL链接地址。

```

from urllib.request import urlopen

class UrlTemplate:
    def __init__(self, template):
        self.template = template

    def open(self, **kwargs):
        return urlopen(self.template.format_map(kwargs))

# Example use. Download stock data from yahoo
yahoo = UrlTemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')
for line in yahoo.open(names='IBM,AAPL,FB', fields='s11c1v'):
    print(line.decode('utf-8'))

```

这个类可以被一个更简单的函数来代替：

```

def urltemplate(template):
    def opener(**kwargs):
        return urlopen(template.format_map(kwargs))
    return opener

# Example use
yahoo = urltemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')
for line in yahoo(names='IBM,AAPL,FB', fields='s11c1v'):
    print(line.decode('utf-8'))

```

讨论

大部分情况下，你拥有一个单方法类的原因是需要存储某些额外的状态来给方法使用。比如，定义 `UrlTemplate` 类的唯一目的就是先在某个地方存储模板值，以便将来可以在 `open()` 方法中使用。

使用一个内部函数或者闭包的方案通常会更优雅一些。简单来讲，一个闭包就是一个函数，只不过在函数内部带上了一个额外的变量环境。闭包关键特点就是它会记住自己被定义时的环境。因此，在我们的解决方案中，`opener()` 函数记住了 `template` 参数的值，并在接下来的调用中使用它。

任何时候只要你碰到需要给某个函数增加额外的状态信息的问题，都可以考虑使用闭包。相比将你的函数转换成一个类而言，闭包通常是一种更加简洁和优雅的方案。

7.10 带额外状态信息的回调函数

问题

你的代码中需要依赖到回调函数的使用(比如事件处理器、等待后台任务完成后的回调等)，并且你还需要让回调函数拥有额外的状态值，以便在它的内部使用到。

解决方案

这一小节主要讨论的是那些出现在很多函数库和框架中的回调函数的使用——特别是跟异步处理有关的。为了演示与测试，我们先定义如下一个需要调用回调函数的函数：

```
def apply_async(func, args, *, callback):  
    # Compute the result  
    result = func(*args)  
  
    # Invoke the callback with the result  
    callback(result)
```

实际上，这段代码可以做任何更高级的处理，包括线程、进程和定时器，但是这些都不是我们要关心的。我们仅仅只需要关注回调函数的调用。下面是一个演示怎样使用上述代码的例子：

```
>>> def print_result(result):  
...     print('Got:', result)  
...  
>>> def add(x, y):  
...     return x + y  
...  
>>> apply_async(add, (2, 3), callback=print_result)  
Got: 5  
>>> apply_async(add, ('hello', 'world'), callback=print_result)  
Got: helloworld  
>>>
```

注意到 `print_result()` 函数仅仅只接受一个参数 `result`。不能再传入其他信息。而当你想让回调函数访问其他变量或者特定环境的变量值的时候就会遇到麻烦。

为了让回调函数访问外部信息，一种方法是使用一个绑定方法来代替一个简单函数。比如，下面这个类会保存一个内部序列号，每次接收到一个 `result` 的时候序列号加1：

```

class ResultHandler:

    def __init__(self):
        self.sequence = 0

    def handler(self, result):
        self.sequence += 1
        print('[{}] Got: {}'.format(self.sequence, result))

```

使用这个类的时候，你先创建一个类的实例，然后用它的 `handler()` 绑定方法来做为回调函数：

```

>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Got: helloworld
>>>

```

第二种方式，作为类的替代，可以使用一个闭包捕获状态值，例如：

```

def make_handler():
    sequence = 0
    def handler(result):
        nonlocal sequence
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
    return handler

```

下面是使用闭包方式的一个例子：

```

>>> handler = make_handler()
>>> apply_async(add, (2, 3), callback=handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler)
[2] Got: helloworld
>>>

```

还有另外一个更高级的方法，可以使用协程来完成同样的事情：

```
def make_handler():
    sequence = 0
    while True:
        result = yield
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
```

对于协程，你需要使用它的 `send()` 方法作为回调函数，如下所示：

```
>>> handler = make_handler()
>>> next(handler) # Advance to the yield
>>> apply_async(add, (2, 3), callback=handler.send)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld
>>>
```

讨论

基于回调函数的软件通常都有可能变得非常复杂。一部分原因是回调函数通常会跟请求执行代码断开。因此，请求执行和处理结果之间的执行环境实际上已经丢失了。如果你想让回调函数连续执行多步操作，那你就必须去解决如何保存和恢复相关的状态信息了。

至少有两种主要方式来捕获和保存状态信息，你可以在一个对象实例(通过一个绑定方法)或者在一个闭包中保存它。两种方式相比，闭包或许是更加轻量级和自然一点，因为它们可以很简单的通过函数来构造。它们还能自动捕获所有被使用到的变量。因此，你无需去担心如何去存储额外的状态信息(代码中自动判定)。

如果使用闭包，你需要注意对那些可修改变量的操作。在上面的方案中，`nonlocal` 声明语句用来指示接下来的变量会在回调函数中被修改。如果没有这个声明，代码会报错。

而使用一个协程来作为一个回调函数就更有意思了，它跟闭包方法密切相关。某种意义上讲，它显得更加简洁，因为总共就一个函数而已。并且，你可以很自由的修改变量而无需去使用 `nonlocal` 声明。这种方式唯一缺点就是相对于其他Python技术而已或许比较难以理解。另外还有一些比较难懂的部分，比如使用之前需要调用 `next()`，实际使用时这个步骤很容易被忘记。尽管如此，协程还有其他用处，比如作为一个内联回调函数的定义(下一节会讲到)。

如果你仅仅只需要给回调函数传递额外的值的话，还有一种使用 `partial()` 的方式也很有用。在没有使用 `partial()` 的时候，你可能经常看到下面这种使用lambda表达式的复杂代码：

```
>>> apply_async(add, (2, 3), callback=lambda r: handler(r, seq))
[1] Got: 5
>>>
```

可以参考7.8小节的几个示例，教你如何使用 `partial()` 来更改参数签名来简化上述代码。

7.11 内联回调函数

问题

当你编写使用回调函数的代码的时候，担心很多小函数的扩张可能会弄乱程序控制流。你希望找到某个方法来让代码看上去更像是一个普通的执行序列。

解决方案

通过使用生成器和协程可以使得回调函数内联在某个函数中。为了演示说明，假设你有如下所示的一个执行某种计算任务然后调用一个回调函数的函数(参考7.10小节)：

```
def apply_async(func, args, *, callback):
    # Compute the result
    result = func(*args)

    # Invoke the callback with the result
    callback(result)
```

接下来让我们看一下下面的代码，它包含了一个 `Async` 类和一个 `inlined_async` 装饰器：

```

from queue import Queue
from functools import wraps

class Async:
    def __init__(self, func, args):
        self.func = func
        self.args = args

def inlined_async(func):
    @wraps(func)
    def wrapper(*args):
        f = func(*args)
        result_queue = Queue()
        result_queue.put(None)
        while True:
            result = result_queue.get()
            try:
                a = f.send(result)
                apply_async(a.func, a.args, callback=result_queue.put)
            except StopIteration:
                break
        return wrapper

```

这两个代码片段允许你使用 `yield` 语句内联回调步骤。比如：

```

def add(x, y):
    return x + y

@inlined_async
def test():
    r = yield Async(add, (2, 3))
    print(r)
    r = yield Async(add, ('hello', 'world'))
    print(r)
    for n in range(10):
        r = yield Async(add, (n, n))
        print(r)
    print('Goodbye')

```

如果你调用 `test()`，你会得到类似如下的输出：

```
5
helloworld
0
2
4
6
8
10
12
14
16
18
Goodbye
```

你会发现，除了那个特别的装饰器和 `yield` 语句外，其他地方并没有出现任何的回调函数(其实是在后台定义的)。

讨论

本小节会实实在在的测试你关于回调函数、生成器和控制流的知识。

首先，在需要使用到回调的代码中，关键点在于当前计算工作会挂起并在将来的某个时候重启(比如异步执行)。当计算重启时，回调函数被调用来继续处理结果。`apply_async()` 函数演示了执行回调的实际逻辑，尽管实际情况中它可能会更加复杂(包括线程、进程、事件处理器等等)。

计算的暂停与重启思路跟生成器函数的执行模型不谋而合。具体来讲，`yield` 操作会使一个生成器函数产生一个值并暂停。接下来调用生成器的 `__next__()` 或 `send()` 方法又会让它从暂停处继续执行。

根据这个思路，这一小节的核心就在 `inline_async()` 装饰器函数中了。关键点就是，装饰器会逐步遍历生成器函数的所有 `yield` 语句，每一次一个。为了这样做，刚开始的时候创建了一个 `result` 队列并向里面放入一个 `None` 值。然后开始一个循环操作，从队列中取出结果值并发送给生成器，它会持续到下一个 `yield` 语句，在这里一个 `Async` 的实例被接受到。然后循环开始检查函数和参数，并开始进行异步计算 `apply_async()`。然而，这个计算有个最诡异部分是它并没有使用一个普通的回调函数，而是用队列的 `put()` 方法来回调。

这时候，是时候详细解释下到底发生了什么事了。主循环立即返回顶部并在队列上执行 `get()` 操作。如果数据存在，它一定是 `put()` 回调存放的结果。如果没有数据，那么先暂停操作并等待结果的到来。这个具体怎样实现是由 `apply_async()` 函数来决定的。如

如果你不相信会有这么神奇的事情，你可以使用 `multiprocessing` 库来试一下，在单独的进程中执行异步计算操作，如下所示：

```
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
    apply_async = pool.apply_async

    # Run the test function
    test()
```

实际上你会发现这个真的就是这样的，但是要解释清楚具体的控制流得需要点时间了。

将复杂的控制流隐藏到生成器函数背后的例子在标准库和第三方包中都能看到。比如，在 `contextlib` 中的 `@contextmanager` 装饰器使用了一个令人费解的技巧，通过一个 `yield` 语句将进入和离开上下文管理器粘合在一起。另外非常流行的 `Twisted` 包中也包含了非常类似的内联回调。

7.12 访问闭包中定义的变量

问题

你想要扩展函数中的某个闭包，允许它能访问和修改函数的内部变量。

解决方案

通常来讲，闭包的内部变量对于外界来讲是完全隐藏的。但是，你可以通过编写访问函数并将其作为函数属性绑定到闭包上来实现这个目的。例如：

```
def sample():
    n = 0
    # Closure function
    def func():
        print('n=', n)

    # Accessor methods for n
    def get_n():
        return n

    def set_n(value):
        nonlocal n
        n = value

    # Attach as function attributes
    func.get_n = get_n
    func.set_n = set_n
    return func
```

下面是使用的例子:

```
>>> f = sample()
>>> f()
n= 0
>>> f.set_n(10)
>>> f()
n= 10
>>> f.get_n()
10
>>>
```

讨论

为了说明清楚它如何工作的，有两点需要解释一下。首先，`nonlocal` 声明可以让我们编写函数来修改内部变量的值。其次，函数属性允许我们用一种很简单的方式将访问方法绑定到闭包函数上，这个跟实例方法很像(尽管并没有定义任何类)。

还可以进一步的扩展，让闭包模拟类的实例。你要做的仅仅是复制上面的内部函数到一个字典实例中并返回它即可。例如：


```

import sys
class ClosureInstance:
    def __init__(self, locals=None):
        if locals is None:
            locals = sys._getframe(1).f_locals

        # Update instance dictionary with callables
        self.__dict__.update((key,value) for key, value in locals.items()
                             if callable(value) )

    # Redirect special methods
    def __len__(self):
        return self.__dict__['__len__']()

# Example use
def Stack():
    items = []
    def push(item):
        items.append(item)

    def pop():
        return items.pop()

    def __len__():
        return len(items)

    return ClosureInstance()

```

下面是一个交互式会话来演示它是如何工作的：

```

>>> s = Stack()
>>> s
<__main__.ClosureInstance object at 0x10069ed10>
>>> s.push(10)
>>> s.push(20)
>>> s.push('Hello')
>>> len(s)
3
>>> s.pop()
'Hello'
>>> s.pop()
20
>>> s.pop()
10
>>>

```

有趣的是，这个代码运行起来会比一个普通的类定义要快很多。你可能会像下面这样测试它跟一个类的性能对比：

```
class Stack2:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def __len__(self):
        return len(self.items)
```

如果这样做，你会得到类似如下的结果：

```
>>> from timeit import timeit
>>> # Test involving closures
>>> s = Stack()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
0.9874754269840196
>>> # Test involving a class
>>> s = Stack2()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
1.0707052160287276
>>>
```

结果显示，闭包的方案运行起来要快大概8%，大部分原因是因为对实例变量的简化访问，闭包更快是因为不会涉及到额外的self变量。

Raymond Hettinger对于这个问题设计出了更加难以理解的改进方案。不过，你得考虑下是否真的需要在你代码中这样做，而且它只是真实类的一个奇怪的替换而已，例如，类的主要特性如继承、属性、描述器或类方法都是不能用的。并且你要做一些其他的工作才能让一些特殊方法生效(比如上面 `ClosureInstance` 中重写过的 `__len__()` 实现。)

最后，你可能还会让其他阅读你代码的人感到疑惑，为什么它看起来不像一个普通的类定义呢？(当然，他们也想知道为什么它运行起来会更快)。尽管如此，这对于怎样访问闭包的内部变量也不失为一个有趣的例子。

总体上讲，在配置的时候给闭包添加方法会有更多的实用功能，比如你需要重置内部状态、刷新缓冲区、清除缓存或其他的反馈机制的时候。

第八章：类与对象

本章主要关注点的是和类定义有关的常见编程模型。包括让对象支持常见的Python特性、特殊方法的使用、类封装技术、继承、内存管理以及有用的设计模式。

Contents:

8.1 改变对象的字符串显示

问题

你想改变对象实例的打印或显示输出，让它们更具可读性。

解决方案

要改变一个实例的字符串表示，可重新定义它的 `__str__()` 和 `__repr__()` 方法。例如：

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)

    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)
```

`__repr__()` 方法返回一个实例的代码表示形式，通常用来重新构造这个实例。内置的 `repr()` 函数返回这个字符串，跟我们使用交互式解释器显示的值是一样的。`__str__()` 方法将实例转换为一个字符串，使用 `str()` 或 `print()` 函数会输出这个字符串。比如：

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4) # __repr__() output
>>> print(p)
(3, 4) # __str__() output
>>>
```

我们在这里还演示了在格式化的时候怎样使用不同的字符串表现形式。特别来讲，`!r` 格式化代码指明输出使用 `__repr__()` 来代替默认的 `__str__()`。你可以用前面的类来试着测试下：

```
>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
p is (3, 4)
>>>
```

讨论

自定义 `__repr__()` 和 `__str__()` 通常是很好的习惯，因为它能简化调试和实例输出。例如，如果仅仅只是打印输出或日志输出某个实例，那么程序员会看到实例更加详细与有用的信息。

`__repr__()` 生成的文本字符串标准做法是需要让 `eval(repr(x)) == x` 为真。如果实在不能这样子做，应该创建一个有用的文本表示，并使用 `<` 和 `>` 括起来。比如：

```
>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>
```

如果 `__str__()` 没有被定义，那么就会使用 `__repr__()` 来代替输出。

上面的 `format()` 方法的使用看上去很有趣，格式化代码 `{0.x}` 对应的是第1个参数的x属性。因此，在下面的函数中，`0`实际上指的就是 `self` 本身：

```
def __repr__(self):
    return 'Pair({0.x!r}, {0.y!r})'.format(self)
```

作为这种实现的一个替代，你也可以使用 `%` 操作符，就像下面这样：

```
def __repr__(self):
    return 'Pair(%r, %r)' % (self.x, self.y)
```

8.2 自定义字符串的格式化

问题

你想通过 `format()` 函数和字符串方法使得一个对象能支持自定义的格式化。

解决方案

为了自定义字符串的格式化，我们需要在类上面定义 `__format__()` 方法。例如：

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
}

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d=self)
```

现在 `Date` 类的实例可以支持格式化操作了，如同下面这样：

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

讨论

`__format__()` 方法给Python的字符串格式化功能提供了一个钩子。这里需要着重强调的是格式化代码的解析工作完全由类自己决定。因此，格式化代码可以是任何值。例如，参考下面来自 `datetime` 模块中的代码：

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, '%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:%d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>
```

对于内置类型的格式化有一些标准的约定。可以参考 [string模块文档](#) 说明。

8.3 让对象支持上下文管理协议

问题

你想让你的对象支持上下文管理协议(with语句)。

解决方案

为了让一个对象兼容 `with` 语句，你需要实现 `__enter__()` 和 `__exit__()` 方法。例如，考虑如下的一个类，它能为我们创建一个网络连接：

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = family
        self.type = type
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

这个类的关键特点在于它表示了一个网络连接，但是初始化的时候并不会做任何事情(比如它并没有建立一个连接)。连接的建立和关闭是使用 `with` 语句自动完成的，例如：

```
from functools import partial

conn = LazyConnection(('www.python.org', 80))
# Connection closed
with conn as s:
    # conn.__enter__() executes: connection open
    s.send(b'GET /index.html HTTP/1.0\r\n')
    s.send(b'Host: www.python.org\r\n')
    s.send(b'\r\n')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() executes: connection closed
```

讨论

编写上下文管理器的主要原理是你的代码会放到 `with` 语句块中执行。当出现 `with` 语句的时候，对象的 `__enter__()` 方法被触发，它返回的值(如果有的话)会被赋值给 `as` 声明的变量。然后，`with` 语句块里面的代码开始执行。最后，`__exit__()` 方法被触发进行清理工作。

不管 `with` 代码块中发生什么，上面的控制流都会执行完，就算代码块中发生了异常也是一样的。事实上，`__exit__()` 方法的第三个参数包含了异常类型、异常值和追溯信息(如果有的话)。`__exit__()` 方法能自己决定怎样利用这个异常信息，或者忽略它并返回一个 `None` 值。如果 `__exit__()` 返回 `True`，那么异常会被清空，就好像什么都没发生一样，`with` 语句后面的程序继续在正常执行。

还有一个细节问题就是 `LazyConnection` 类是否允许多个 `with` 语句来嵌套使用连接。很显然，上面的定义中一次只能允许一个 `socket` 连接，如果正在使用一个 `socket` 的时候又重复使用 `with` 语句，就会产生一个异常了。不过你可以像下面这样修改下上面的实现来解决这个问题：

```

from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = family
        self.type = type
        self.connections = []

    def __enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        self.connections.append(sock)
        return sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.connections.pop().close()

# Example use
from functools import partial

conn = LazyConnection(('www.python.org', 80))
with conn as s1:
    pass
    with conn as s2:
        pass
        # s1 and s2 are independent sockets

```

在第二个版本中，`LazyConnection` 类可以被看做是某个连接工厂。在内部，一个列表被用来构造一个栈。每次 `__enter__()` 方法执行的时候，它复制创建一个新的连接并将其加入到栈里面。`__exit__()` 方法简单的从栈中弹出最后一个连接并关闭它。这里稍微有点难理解，不过它能允许嵌套使用 `with` 语句创建多个连接，就如上面演示的那样。

在需要管理一些资源比如文件、网络连接和锁的编程环境中，使用上下文管理器是很普遍的。这些资源的一个主要特征是它们必须被手动的关闭或释放来确保程序的正确运行。例如，如果你请求了一个锁，那么你必须确保之后释放了它，否则就可能产生死锁。通过实现 `__enter__()` 和 `__exit__()` 方法并使用 `with` 语句可以很容易的避免这些问题，因为 `__exit__()` 方法可以让你无需担心这些了。

在 `contextmanager` 模块中有一个标准的上下文管理方案模板，可参考9.22小节。同时在12.6小节中还有一个对本节示例程序的线程安全的修改版。

8.4 创建大量对象时节省内存方法

问题

你的程序要创建大量(可能上百万)的对象，导致占用很大的内存。

解决方案

对于主要是用来当成简单的数据结构的类而言，你可以通过给类添加 `__slots__` 属性来极大的减少实例所占的内存。比如：

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

当你定义 `__slots__` 后，Python就会为实例使用一种更加紧凑的内部表示。实例通过一个很小的固定大小的数组来构建，而不是为每个实例定义一个字典，这跟元组或列表很类似。在 `__slots__` 中列出的属性名在内部被映射到这个数组的指定小标上。使用slots一个不好的地方就是我们不能再给实例添加新的属性了，只能使用在 `__slots__` 中定义的那些属性名。

讨论

使用slots后节省的内存会跟存储属性的数量和类型有关。不过，一般来讲，使用到的内存总量和将数据存储在一个元组中差不多。为了给你一个直观认识，假设你不使用slots直接存储一个Date实例，在64位的Python上面要占用428字节，而如果使用了slots，内存占用下降到156字节。如果程序中需要同时创建大量的日期实例，那么这个就能极大的减小内存使用量了。

尽管slots看上去是一个很有用的特性，很多时候你还是得减少对它的使用冲动。Python的很多特性都依赖于普通的基于字典的实现。另外，定义了slots后的类不再支持一些普通类特性了，比如多继承。大多数情况下，你应该只在那些经常被使用到的用作数据结构的类上定义slots (比如在程序中需要创建某个类的几百万个实例对象)。

关于 `__slots__` 的一个常见误区是它可以作为一个封装工具来防止用户给实例增加新的属性。尽管使用slots可以达到这样的目的，但是这个并不是它的初衷。`__slots__` 更多的是用来作为一个内存优化工具。

8.5 在类中封装属性名

问题

你想封装类的实例上面的“私有”数据，但是Python语言并没有访问控制。

解决方案

Python程序员不去依赖语言特性去封装数据，而是通过遵循一定的属性和方法命名规约来达到这个效果。第一个约定是任何以单下划线_开头的名字都应该是内部实现。比如：

```
class A:
    def __init__(self):
        self._internal = 0 # An internal attribute
        self.public = 1 # A public attribute

    def public_method(self):
        ...

        A public method
        ...

    pass

    def _internal_method(self):
        pass
```

Python并不会真的阻止别人访问内部名称。但是如果你这么做肯定是不好的，可能会导致脆弱的代码。同时还要注意到，使用下划线开头的约定同样适用于模块名和模块级别函数。例如，如果你看到某个模块名以单下划线开头(比如_socket)，那它就是内部实现。类似的，模块级别函数比如 `sys._getframe()` 在使用的时候就得加倍小心了。

你还可能会遇到在类定义中使用两个下划线(__)开头的命名。比如：

```
class B:
    def __init__(self):
        self.__private = 0

    def __private_method(self):
        pass

    def public_method(self):
        pass
        self.__private_method()
```

使用双下划线开始会导致访问名称变成其他形式。比如，在前面的类B中，私有属性会被分别重命名为 `_B__private` 和 `_B__private_method`。这时候你可能会问这样重命名的目的是什么，答案就是继承——这种属性通过继承是无法被覆盖的。比如：

```
class C(B):
    def __init__(self):
        super().__init__()
        self.__private = 1 # Does not override B.__private

    # Does not override B.__private_method()
    def __private_method(self):
        pass
```

这里，私有名称 `__private` 和 `__private_method` 被重命名为 `_C__private` 和 `_C__private_method`，这个跟父类B中的名称是完全不同的。

讨论

上面提到有两种不同的编码约定(单下划线和双下划线)来命名私有属性，那么问题就来了：到底哪种方式好呢？大多数而言，你应该让你的非公共名称以单下划线开头。但是，如果你清楚你的代码会涉及到子类，并且有些内部属性应该在子类中隐藏起来，那么才考虑使用双下划线方案。

还有一点要注意的是，有时候你定义的一个变量和某个保留关键字冲突，这时候可以使用单下划线作为后缀，例如：

```
lambda_ = 2.0 # Trailing _ to avoid clash with lambda keyword
```

这里我们并不使用单下划线前缀的原因是它避免误解它的使用初衷(如使用单下划线前缀的目的是为了防止命名冲突而不是指明这个属性是私有的)。通过使用单下划线后缀可以解决这个问题。

8.6 创建可管理的属性

问题

你想给某个实例attribute增加除访问与修改之外的其他处理逻辑，比如类型检查或合法性验证。

解决方案

自定义某个属性的一种简单方法是将其定义为一个property。例如，下面的代码定义了一个property，增加对一个属性简单的类型检查：

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    # Getter function
    @property
    def first_name(self):
        return self._first_name

    # Setter function
    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    @first_name.deleter
    def first_name(self):
        raise AttributeError("Can't delete attribute")
```

上述代码中有三个相关联的方法，这三个方法的名字都必须一样。第一个方法是一个getter函数，它使得first_name成为一个属性。其他两个方法给first_name属性添加了setter和deleter函数。需要强调的是只有在first_name属性被创建后，后面的两个装饰器@first_name.setter和@first_name.deleter才能被定义。

property的一个关键特征是它看上去跟普通的attribute没什么两样，但是访问它的时候会自动触发getter、setter和deleter方法。例如：

```
>>> a = Person('Guido')
>>> a.first_name # Calls the getter
'Guido'
>>> a.first_name = 42 # Calls the setter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "prop.py", line 14, in first_name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.first_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute
>>>
```

在实现一个property的时候，底层数据(如果有的话)仍然需要存储在某个地方。因此，在get和set方法中，你会看到对 `_first_name` 属性的操作，这也是实际数据保存的地方。另外，你可能还会问为什么 `__init__()` 方法中设置了 `self.first_name` 而不是 `self._first_name`。在这个例子中，我们创建一个property的目的就是在设置attribute的时候进行检查。因此，你可能想在初始化的时候也进行这种类型检查。通过设置 `self.first_name`，自动调用 `setter` 方法，这个方法里面会进行参数的检查，否则就是直接访问 `self._first_name` 了。

还能在已存在的get和set方法基础上定义property。例如：

```
class Person:
    def __init__(self, first_name):
        self.set_first_name(first_name)

    # Getter function
    def get_first_name(self):
        return self._first_name

    # Setter function
    def set_first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    def del_first_name(self):
        raise AttributeError("Can't delete attribute")

    # Make a property from existing get/set methods
    name = property(get_first_name, set_first_name, del_first_name)
```

讨论

一个property属性其实就是一系列相关绑定方法的集合。如果你去查看拥有property的类，就会发现property本身的fget、fset和fdel属性就是类里面的普通方法。比如：

```
>>> Person.first_name.fget
<function Person.first_name at 0x1006a60e0>
>>> Person.first_name.fset
<function Person.first_name at 0x1006a6170>
>>> Person.first_name.fdel
<function Person.first_name at 0x1006a62e0>
>>>
```

通常来讲，你不会直接取调用`fget`或者`fset`，它们会在访问`property`的时候自动被触发。

只有当你确实需要对`attribute`执行其他额外的操作的时候才应该使用到`property`。有时候一些从其他编程语言(比如`Java`)过来的程序员总认为所有访问都应该通过`getter`和`setter`，所以他们认为代码应该像下面这样写：

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        self._first_name = value
```

不要写这种没有做任何其他额外操作的`property`。首先，它会让你的代码变得很臃肿，并且还会迷惑阅读者。其次，它还会让你的程序运行起来变慢很多。最后，这样的设计并没有带来任何的好处。特别是当你以后想给普通`attribute`访问添加额外的处理逻辑的时候，你可以将它变成一个`property`而无需改变原来的代码。因为访问`attribute`的代码还是保持原样。

`Properties`还是一种定义动态计算`attribute`的方法。这种类型的`attributes`并不会被实际的存储，而是在需要的时候计算出来。比如：

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return math.pi * self.radius ** 2

    @property
    def diameter(self):
        return self.radius ** 2

    @property
    def perimeter(self):
        return 2 * math.pi * self.radius
```

在这里，我们通过使用`properties`，将所有的访问接口形式统一起来，对半径、直径、周长和面积的访问都是通过属性访问，就跟访问简单的`attribute`是一样的。如果不这样做的话，那么就要在代码中混合使用简单属性访问和方法调用。下面是使用的实例：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area # Notice Lack of ()
50.26548245743669
>>> c.perimeter # Notice Lack of ()
25.132741228718345
>>>
```

尽管`properties`可以实现优雅的编程接口，但有些时候你还是会想直接使用`getter`和`setter`函数。例如：

```
>>> p = Person('Guido')
>>> p.get_first_name()
'Guido'
>>> p.set_first_name('Larry')
>>>
```

这种情况的出现通常是因为Python代码被集成到一个大型基础平台架构或程序中。例如，有可能是一个Python类准备加入到一个基于远程过程调用的大型分布式系统中。这种情况下，直接使用`get/set`方法(普通方法调用)而不是`property`或许会更容易兼容。

最后一点，不要像下面这样写有大量重复代码的`property`定义：

```

class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Repeated property code, but for a different name (bad!)
    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._last_name = value

```

重复代码会导致臃肿、易出错和丑陋的程序。好消息是，通过使用装饰器或闭包，有很多种更好的方法来完成同样的事情。可以参考8.9和9.21小节的内容。

8.7 调用父类方法

问题

你想在子类中调用父类的某个已经被覆盖的方法。

解决方案

为了调用父类(超类)的一个方法，可以使用 `super()` 函数，比如：

```

class A:
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')
        super().spam() # Call parent spam()

```


`super()` 函数的一个常见用法是在 `__init__()` 方法中确保父类被正确的初始化了:

```
class A:
    def __init__(self):
        self.x = 0

class B(A):
    def __init__(self):
        super().__init__()
        self.y = 1
```

`super()` 的另外一个常见用法出现在覆盖Python特殊方法的代码中, 比如:

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value) # Call original __setattr__
        else:
            setattr(self._obj, name, value)
```

在上面代码中, `__setattr__()` 的实现包含一个名字检查。如果某个属性名以下划线(`_`)开头, 就通过 `super()` 调用原始的 `__setattr__()`, 否则的话就委派给内部的代理对象 `self._obj` 去处理。这看上去有点意思, 因为就算没有显式的指明某个类的父类, `super()` 仍然可以有效的工作。

讨论

实际上, 大家对于在Python中如何正确使用 `super()` 函数普遍知之甚少。你有时候会看到像下面这样直接调用父类的一个方法:

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
```

尽管对于大部分代码而言这么做没什么问题，但是在更复杂的涉及到多继承的代码中就有可能导致很奇怪的问题发生。比如，考虑如下的情况：

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

如果你运行这段代码就会发现 `Base.__init__()` 被调用两次，如下所示：

```
>>> c = C()
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
>>>
```

可能两次调用 `Base.__init__()` 没什么坏处，但有时候却不是。另一方面，假设你在代码中换成使用 `super()`，结果就很完美了：

```

class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A,B):
    def __init__(self):
        super().__init__() # Only one call to super() here
        print('C.__init__')

```

运行这个新版本后，你会发现每个 `__init__()` 方法只会被调用一次了：

```

>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
>>>

```

为了弄清它的原理，我们需要花点时间解释下Python是如何实现继承的。对于你定义的每一个类而已，Python会计算出一个所谓的方法解析顺序(MRO)列表。这个MRO列表就是一个简单的所有基类的线性顺序表。例如：

```

>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class '__main__.Base'>, <class 'object'>)
>>>

```

为了实现继承，Python会在MRO列表上从左到右开始查找基类，直到找到第一个匹配这个属性的类为止。

而这个MRO列表的构造是通过一个C3线性化算法来实现的。我们不去深究这个算法的数学原理，它实际上就是合并所有父类的MRO列表并遵循如下三条准则：

- 子类会先于父类被检查

- 多个父类会根据它们在列表中的顺序被检查
- 如果对下一个类存在两个合法的选择，选择第一个父类

老实说，你所要知道的就是MRO列表中的类顺序会让你定义的任意类层级关系变得有意义。

当你使用 `super()` 函数时，Python会在MRO列表上继续搜索下一个类。只要每个重定义的方法统一使用 `super()` 并只调用它一次，那么控制流最终会遍历完整个MRO列表，每个方法也只會被调用一次。这也是为什么在第二个例子中你不会调用两次 `Base.__init__()` 的原因。

`super()` 有个令人吃惊的地方是它并不一定去查找某个类在MRO中下一个直接父类，你甚至可以在一个没有直接父类的类中使用它。例如，考虑如下这个类：

```
class A:
    def spam(self):
        print('A.spam')
        super().spam()
```

如果你试着直接使用这个类就会出错：

```
>>> a = A()
>>> a.spam()
A.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in spam
AttributeError: 'super' object has no attribute 'spam'
>>>
```

但是，如果你使用多继承的话看看会发生什么：

```
>>> class B:
...     def spam(self):
...         print('B.spam')
...
>>> class C(A,B):
...     pass
...
>>> c = C()
>>> c.spam()
A.spam
B.spam
>>>
```

你可以看到在类A中使用 `super().spam()` 实际上调用的是跟类A毫无关系的类B中的 `spam()` 方法。这个用类C的MRO列表就可以完全解释清楚了：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
 <class 'object'>)
>>>
```

在定义混入类的时候这样使用 `super()` 是很普遍的。可以参考8.13和8.18小节。

然而，由于 `super()` 可能会调用不是你想要的方法，你应该遵循一些通用原则。首先，确保在继承体系中所有相同名字的方法拥有可兼容的参数签名(比如相同的参数个数和参数名称)。这样可以确保 `super()` 调用一个非直接父类方法时不会出错。其次，最好确保最顶层的类提供了这个方法的实现，这样的话在MRO上面的查找链肯定可以找到某个确定的方法。

在Python社区中对于 `super()` 的使用有时候会引来一些争议。尽管如此，如果一切顺利的话，你应该在你最新代码中使用它。Raymond Hettinger为此写了一篇非常好的文章“[Python's super\(\) Considered Super!](#)”，通过大量的例子向我们解释了为什么 `super()` 是极好的。

8.8 子类中扩展property

问题

在子类中，你想要扩展定义在父类中的property的功能。

解决方案

考虑如下的代码，它定义了一个property：

```

class Person:
    def __init__(self, name):
        self.name = name

    # Getter function
    @property
    def name(self):
        return self._name

    # Setter function
    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._name = value

    # Deleter function
    @name.deleter
    def name(self):
        raise AttributeError("Can't delete attribute")

```

下面是一个示例类，它继承自Person并扩展了 `name` 属性的功能：

```

class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)

```

接下来使用这个新类：

```
>>> s = SubPerson('Guido')
Setting name to Guido
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
Setting name to Larry
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

如果你仅仅只想扩展property的某一个方法，那么可以像下面这样写：

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

或者，你只想修改setter方法，就这么写：

```
class SubPerson(Person):
    @Person.name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)
```

讨论

在子类中扩展一个property可能会引起很多不易察觉的问题，因为一个property其实是getter、setter和deleter方法的集合，而不是单个方法。因此，当你扩展一个property的时候，你需要先确定你是否要重新定义所有的方法还是说只修改其中某一个。

在第一个例子中，所有的property方法都被重新定义。在每一个方法中，使用了super()来调用父类的实现。在setter函数中使用

super(SubPerson, SubPerson).name.__set__(self, value)的语句是没有错的。为了委托给

之前定义的setter方法，需要将控制权传递给之前定义的name属性的 `__set__()` 方法。不过，获取这个方法的唯一途径是使用类变量而不是实例变量来访问它。这也是为什么我们要使用 `super(SubPerson, SubPerson)` 的原因。

如果你只想重定义其中一个方法，那只使用 `@property` 本身是不够的。比如，下面的代码就无法工作：

```
class SubPerson(Person):
    @property # Doesn't work
    def name(self):
        print('Getting name')
        return super().name
```

如果你试着运行会发现setter函数整个消失了：

```
>>> s = SubPerson('Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 5, in __init__
    self.name = name
AttributeError: can't set attribute
>>>
```

你应该像之前说过的那样修改代码：

```
class SubPerson(Person):
    @Person.getter
    def name(self):
        print('Getting name')
        return super().name
```

这么写后，`property`之前已经定义过的方法会被复制过来，而`getter`函数被替换。然后它就能按照期望的工作了：


```
>>> s = SubPerson('Guido')
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
>>> s.name
Getting name
'Larry'
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

在这个特别的解决方案中，我们没办法使用更加通用的方式去替换硬编码的 `Person` 类名。如果你不知道到底是哪个基类定义了 `property`，那你只能通过重新定义所有 `property` 并使用 `super()` 来将控制权传递给前面的实现。

值得注意的是上面演示的第一种技术还可以被用来扩展一个描述器(在8.9小节我们有专门的介绍)。比如：

```

# A descriptor
class String:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        instance.__dict__[self.name] = value

# A class with a descriptor
class Person:
    name = String('name')

    def __init__(self, name):
        self.name = name

# Extending a descriptor with a property
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)

```

最后值得注意的是，读到这里时，你应该会发现子类化 `setter` 和 `deleter` 方法其实是很简单的。这里演示的解决方案同样适用，但是在 [Python的issue页面](#) 报告的一个bug，或许会使得将来的Python版本中出现一个更加简洁的方法。

8.9 创建新的类或实例属性

问题

你想创建一个新的拥有一些额外功能的实例属性类型，比如类型检查。

解决方案

如果你想创建一个全新的实例属性，可以通过一个描述器类的形式来定义它的功能。下面是一个例子：

```
# Descriptor attribute for an integer type-checked attribute
class Integer:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
```

一个描述器就是一个实现了三个核心的属性访问操作(`get`, `set`, `delete`)的类，分别为 `__get__()`、`__set__()` 和 `__delete__()` 这三个特殊的方法。这些方法接受一个实例作为输入，之后相应的操作实例底层的字典。

为了使用一个描述器，需将这个描述器的实例作为类属性放到一个类的定义中。例如：

```
class Point:
    x = Integer('x')
    y = Integer('y')

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

当你这样做后，所有对描述器属性(比如`x`或`y`)的访问会被 `__get__()`、`__set__()` 和 `__delete__()` 方法捕获到。例如：

```
>>> p = Point(2, 3)
>>> p.x # Calls Point.x.__get__(p,Point)
2
>>> p.y = 5 # Calls Point.y.__set__(p, 5)
>>> p.x = 2.3 # Calls Point.x.__set__(p, 2.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "descrip.py", line 12, in __set__
    raise TypeError('Expected an int')
TypeError: Expected an int
>>>
```

作为输入，描述器的每一个方法会接受一个操作实例。为了实现请求操作，会相应的操作实例底层的字典(`__dict__`属性)。描述器的 `self.name` 属性存储了在实例字典中被实际使用到的key。

讨论

描述器可实现大部分Python类特性中的底层魔法，包括 `@classmethod`、`@staticmethod`、`@property`，甚至是 `__slots__` 特性。

通过定义一个描述器，你可以在底层捕获核心的实例操作(`get, set, delete`)，并且可完全自定义它们的行为。这是一个强大的工具，有了它你可以实现很多高级功能，并且它也是很多高级库和框架中的重要工具之一。

描述器的一个比较困惑的地方是它只能在类级别被定义，而不能为每个实例单独定义。因此，下面的代码是无法工作的：

```
# Does NOT work
class Point:
    def __init__(self, x, y):
        self.x = Integer('x') # No! Must be a class variable
        self.y = Integer('y')
        self.x = x
        self.y = y
```

同时，`__get__()` 方法实现起来比看上去要复杂得多：

```
# Descriptor attribute for an integer type-checked attribute
class Integer:

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
```

`__get__()` 看上去有点复杂的原因归结于实例变量和类变量的不同。如果一个描述器被当作一个类变量来访问，那么 `instance` 参数被设置成 `None`。这种情况下，标准做法就是简单的返回这个描述器本身即可(尽管你还可以添加其他的自定义操作)。例如：

```
>>> p = Point(2,3)
>>> p.x # Calls Point.x.__get__(p, Point)
2
>>> Point.x # Calls Point.x.__get__(None, Point)
<__main__.Integer object at 0x100671890>
>>>
```

描述器通常是那些使用到装饰器或元类的大型框架中的一个组件。同时它们的使用也被隐藏在后面。举个例子，下面是一些更高级的基于描述器的代码，并涉及到一个类装饰器：

```

# Descriptor for a type-checked attribute
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value
    def __delete__(self, instance):
        del instance.__dict__[self.name]

# Class decorator that applies it to selected attributes
def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            # Attach a Typed descriptor to the class
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate

# Example use
@typeassert(name=str, shares=int, price=float)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

```

最后要指出的一点是，如果你只是想简单的自定义某个类的单个属性访问的话就不用去写描述器了。这种情况下使用8.6小节介绍的property技术会更加容易。当程序中有很多重复代码的时候描述器就很有用了(比如你想在你代码的很多地方使用描述器提供的功能或者将它作为一个函数库特性)。

8.10 使用延迟计算属性

问题

你想将一个只读属性定义成一个property，并且只在访问的时候才会计算结果。但是一旦被访问后，你希望结果值被缓存起来，不用每次都去计算。

解决方案

定义一个延迟属性的一种高效方法是通过使用一个描述器类，如下所示：

```
class lazyproperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            value = self.func(instance)
            setattr(instance, self.func.__name__, value)
            return value
```

你需要像下面这样在一个类中使用它：

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @lazyproperty
    def area(self):
        print('Computing area')
        return math.pi * self.radius ** 2

    @lazyproperty
    def perimeter(self):
        print('Computing perimeter')
        return 2 * math.pi * self.radius
```

下面在一个交互环境中演示它的使用：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.perimeter
Computing perimeter
25.132741228718345
>>> c.perimeter
25.132741228718345
>>>
```

仔细观察你会发现消息 `Computing area` 和 `Computing perimeter` 仅仅出现一次。

讨论

很多时候，构造一个延迟计算属性的主要目的是为了提升性能。例如，你可以避免计算这些属性值，除非你真的需要它们。这里演示的方案就是用来实现这样的效果的，只不过它是通过以非常高效的方式使用描述器的一个精妙特性来达到这种效果的。

正如在其他小节(如8.9小节)所讲的那样，当一个描述器被放入一个类的定义时，每次访问属性时它的 `__get__()`、`__set__()` 和 `__delete__()` 方法就会被触发。不过，如果一个描述器仅仅只定义了一个 `__get__()` 方法的话，它比通常的具有更弱的绑定。特别地，只有当被访问属性不在实例底层的字典中时 `__get__()` 方法才会被触发。

`lazyproperty` 类利用这一点，使用 `__get__()` 方法在实例中存储计算出来的值，这个实例使用相同的名字作为它的property。这样一来，结果值被存储在实例字典中并且以后就不需要再去计算这个property了。你可以尝试更深入的例子来观察结果：

```
>>> c = Circle(4.0)
>>> # Get instance variables
>>> vars(c)
{'radius': 4.0}

>>> # Compute area and observe variables afterward
>>> c.area
Computing area
50.26548245743669
>>> vars(c)
{'area': 50.26548245743669, 'radius': 4.0}

>>> # Notice access doesn't invoke property anymore
>>> c.area
50.26548245743669

>>> # Delete the variable and see property trigger again
>>> del c.area
>>> vars(c)
{'radius': 4.0}
>>> c.area
Computing area
50.26548245743669
>>>
```

这种方案有一个小缺陷就是计算出的值被创建后是可以被修改的。例如：


```
>>> c.area
Computing area
50.26548245743669
>>> c.area = 25
>>> c.area
25
>>>
```

如果你担心这个问题，那么可以使用一种稍微没那么高效的实现，就像下面这样：

```
def lazyproperty(func):
    name = '_lazy_' + func.__name__
    @property
    def lazy(self):
        if hasattr(self, name):
            return getattr(self, name)
        else:
            value = func(self)
            setattr(self, name, value)
            return value
    return lazy
```

如果你使用这个版本，就会发现现在修改操作已经不被允许了：

```
>>> c = Circle(4.0)
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.area = 25
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

然而，这种方案有一个缺点就是所有`get`操作都必须被定向到属性的 `getter` 函数上去。这个跟之前简单的在实例字典中查找值的方案相比效率要低一点。如果想获取更多关于 `property` 和可管理属性的信息，可以参考8.6小节。而描述器的相关内容可以在8.9小节找到。

8.11 简化数据结构的初始化

问题

你写了很多仅仅用作数据结构的类，不想写太多烦人的 `__init__()` 函数

解决方案

可以在一个基类中写一个公用的 `__init__()` 函数：

```
import math

class Structure1:
    # Class variable that specifies expected fields
    _fields = []

    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))
        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)
```

然后使你的类继承自这个基类：

```
# Example class definitions
class Stock(Structure1):
    _fields = ['name', 'shares', 'price']

class Point(Structure1):
    _fields = ['x', 'y']

class Circle(Structure1):
    _fields = ['radius']

    def area(self):
        return math.pi * self.radius ** 2
```

使用这些类的示例：

```
>>> s = Stock('ACME', 50, 91.1)
>>> p = Point(2, 3)
>>> c = Circle(4.5)
>>> s2 = Stock('ACME', 50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "structure.py", line 6, in __init__
    raise TypeError('Expected {} arguments'.format(len(self._fields)))
TypeError: Expected 3 arguments
```

如果还想支持关键字参数，可以将关键字参数设置为实例属性：

```
class Structure2:
    _fields = []

    def __init__(self, *args, **kwargs):
        if len(args) > len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set all of the positional arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the remaining keyword arguments
        for name in self._fields[len(args):]:
            setattr(self, name, kwargs.pop(name))

        # Check for any remaining unknown arguments
        if kwargs:
            raise TypeError('Invalid argument(s): {}'.format(','.join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure2):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, price=91.1)
    s3 = Stock('ACME', shares=50, price=91.1)
    # s3 = Stock('ACME', shares=50, price=91.1, aa=1)
```

你还能将不在 `_fields` 中的名称加入到属性中去：

```

class Structure3:
    # Class variable that specifies expected fields
    _fields = []

    def __init__(self, *args, **kwargs):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the additional arguments (if any)
        extra_args = kwargs.keys() - self._fields
        for name in extra_args:
            setattr(self, name, kwargs.pop(name))

        if kwargs:
            raise TypeError('Duplicate values for {}'.format(','.join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure3):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, 91.1, date='8/2/2012')

```

讨论

当你需要使用大量很小的数据结构类的时候，相比手工一个个定义 `__init__()` 方法而已，使用这种方式可以大大简化代码。

在上面的实现中我们使用了 `setattr()` 函数类设置属性值，你可能不想用这种方式，而是想直接更新实例字典，就像下面这样：

```

class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments (alternate)
        self.__dict__.update(zip(self._fields,args))

```

尽管这也可以正常工作，但是当定义子类的时候问题就来了。当一个子类定义了 `__slots__` 或者通过 `property`(或描述器)来包装某个属性，那么直接访问实例字典就不起作用了。我们上面使用 `setattr()` 会显得更通用些，因为它也适用于子类情况。

这种方法唯一不好的地方就是对某些IDE而已，在显示帮助函数时可能不太友好。比如：

```
>>> help(Stock)
Help on class Stock in module __main__:
class Stock(Structure)
...
| Methods inherited from Structure:
|
| __init__(self, *args, **kwargs)
|
...
>>>
```

可以参考9.16小节来强制在 `__init__()` 方法中指定参数的类型签名。

8.12 定义接口或者抽象基类

问题

你想定义一个接口或抽象类，并且通过执行类型检查来确保子类实现了某些特定的方法

解决方案

使用 `abc` 模块可以很轻松的定义抽象基类：

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxbytes=-1):
        pass

    @abstractmethod
    def write(self, data):
        pass
```

抽象类的一个特点是它不能被实例化，比如你想像下面这样做是不行的：

```
a = IStream() # TypeError: Can't instantiate abstract class
              # IStream with abstract methods read, write
```

抽象类的目的就是让别的类继承它并实现特定的抽象方法：

```
class SocketStream(IStream):
    def read(self, maxbytes=-1):
        pass

    def write(self, data):
        pass
```

抽象基类的一个主要用途是在代码中检查某些类是否为特定类型，实现了特定接口：

```
def serialize(obj, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected an IStream')
    pass
```

除了继承这种方式外，还可以通过注册方式来让某个类实现抽象基类：

```
import io

# Register the built-in I/O classes as supporting our interface
IStream.register(io.IOBase)

# Open a normal file and type check
f = open('foo.txt')
isinstance(f, IStream) # Returns True
```

`@abstractmethod` 还能注解静态方法、类方法和 `properties` 。你只需保证这个注解紧靠在函数定义前即可：

```
class A(metaclass=ABCMeta):
    @property
    @abstractmethod
    def name(self):
        pass

    @name.setter
    @abstractmethod
    def name(self, value):
        pass

    @classmethod
    @abstractmethod
    def method1(cls):
        pass

    @staticmethod
    @abstractmethod
    def method2():
        pass
```

讨论

标准库中有很多用到抽象基类的地方。`collections` 模块定义了很多跟容器和迭代器(序列、映射、集合等)有关的抽象基类。`numbers` 库定义了跟数字对象(整数、浮点数、有理数等)有关的基类。`io` 库定义了很多跟I/O操作相关的基类。

你可以使用预定义的抽象类来执行更通用的类型检查，例如：

```
import collections

# Check if x is a sequence
if isinstance(x, collections.Sequence):
    ...

# Check if x is iterable
if isinstance(x, collections.Iterable):
    ...

# Check if x has a size
if isinstance(x, collections.Sized):
    ...

# Check if x is a mapping
if isinstance(x, collections.Mapping):
```

尽管ABCs可以让我们很方便的做类型检查，但是我们在代码中最好不要过多的使用它。因为Python的本质是一门动态编程语言，其目的就是给你更多灵活性，强制类型检查或让你代码变得更复杂，这样做无异于舍本求末。

8.13 实现数据模型的类型约束

问题

你想定义某些在属性赋值上面有限制的数据结构。

解决方案

在这个问题中，你需要在对某些实例属性赋值时进行检查。所以你要自定义属性赋值函数，这种情况下最好使用描述器。

下面的代码使用描述器实现了一个系统类型和赋值验证框架：


```

# Base class. Uses a descriptor to set a value
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

# Descriptor for enforcing types
class Typed(Descriptor):
    expected_type = type(None)

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('expected ' + str(self.expected_type))
        super().__set__(instance, value)

# Descriptor for enforcing values
class Unsigned(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)

class MaxSized(Descriptor):
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super().__init__(name, **opts)

    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super().__set__(instance, value)

```

这些类就是你要创建的数据模型或类型系统的基础构建模块。下面就是我们实际定义的各种不同的数据类型：

```
class Integer(Typed):
    expected_type = int

class UnsignedInteger(Integer, Unsigned):
    pass

class Float(Typed):
    expected_type = float

class UnsignedFloat(Float, Unsigned):
    pass

class String(Typed):
    expected_type = str

class SizedString(String, MaxSized):
    pass
```

然后使用这些自定义数据类型，我们定义一个类：

```
class Stock:
    # Specify constraints
    name = SizedString('name', size=8)
    shares = UnsignedInteger('shares')
    price = UnsignedFloat('price')

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

然后测试这个类的属性赋值约束，可发现对某些属性的赋值违反了约束是不合法的：

```

>>> s.name
'ACME'
>>> s.shares = 75
>>> s.shares = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 17, in __set__
    super().__set__(instance, value)
  File "example.py", line 23, in __set__
    raise ValueError('Expected >= 0')
ValueError: Expected >= 0
>>> s.price = 'a lot'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in __set__
    raise TypeError('expected ' + str(self.expected_type))
TypeError: expected <class 'float'>
>>> s.name = 'ABRACADABRA'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 17, in __set__
    super().__set__(instance, value)
  File "example.py", line 35, in __set__
    raise ValueError('size must be < ' + str(self.size))
ValueError: size must be < 8
>>>

```

还有一些技术可以简化上面的代码，其中一种是使用类装饰器：

```

# Class decorator to apply constraints
def check_attributes(**kwargs):
    def decorate(cls):
        for key, value in kwargs.items():
            if isinstance(value, Descriptor):
                value.name = key
                setattr(cls, key, value)
            else:
                setattr(cls, key, value(key))
        return cls
    return decorate

# Example
@check_attributes(name=SizedString(size=8),
                 shares=UnsignedInteger,
                 price=UnsignedFloat)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

```

另外一种方式是使用元类:

```
# A metaclass that applies checking
class checkedmeta(type):
    def __new__(cls, clsname, bases, methods):
        # Attach attribute names to the descriptors
        for key, value in methods.items():
            if isinstance(value, Descriptor):
                value.name = key
        return type.__new__(cls, clsname, bases, methods)

# Example
class Stock2(metaclass=checkedmeta):
    name = SizedString(size=8)
    shares = UnsignedInteger()
    price = UnsignedFloat()

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

讨论

本节使用了很多高级技术, 包括描述器、混入类、`super()` 的使用、类装饰器和元类。不可能在这里一一详细展开来讲, 但是可以在8.9、8.18、9.19小节找到更多例子。但是, 我在这里还是要提一下几个需要注意的点。

首先, 在 `Descriptor` 基类中你会看到有个 `__set__()` 方法, 却没有相应的 `__get__()` 方法。如果一个描述仅仅是从底层实例字典中获取某个属性值的话, 那么没必要去定义 `__get__()` 方法。

所有描述器类都是基于混入类来实现的。比如 `Unsigned` 和 `MaxSized` 要跟其他继承自 `Typed` 类混入。这里利用多继承来实现相应的功能。

混入类的一个比较难理解的地方是, 调用 `super()` 函数时, 你并不知道究竟要调用哪个具体类。你需要跟其他类结合后才能正确的使用, 也就是必须合作才能产生效果。

使用类装饰器和元类通常可以简化代码。上面两个例子中你会发现你只需要输入一次属性名即可了。

```

# Normal
class Point:
    x = Integer('x')
    y = Integer('y')

# Metaclass
class Point(metaclass=checkedmeta):
    x = Integer()
    y = Integer()

```

所有方法中，类装饰器方案应该是最灵活和最高明的。首先，它并不依赖任何其他新的技术，比如元类。其次，装饰器可以很容易的添加或删除。

最后，装饰器还能作为混入类的替代技术来实现同样的效果；

```

# Decorator for applying type checking
def Typed(expected_type, cls=None):
    if cls is None:
        return lambda cls: Typed(expected_type, cls)
    super_set = cls.__set__

    def __set__(self, instance, value):
        if not isinstance(value, expected_type):
            raise TypeError('expected ' + str(expected_type))
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls

# Decorator for unsigned values
def Unsigned(cls):
    super_set = cls.__set__

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls

# Decorator for allowing sized values
def MaxSized(cls):
    super_init = cls.__init__

    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super_init(self, name, **opts)

```

```

cls.__init__ = __init__

super_set = cls.__set__

def __set__(self, instance, value):
    if len(value) >= self.size:
        raise ValueError('size must be < ' + str(self.size))
    super_set(self, instance, value)

cls.__set__ = __set__
return cls

# Specialized descriptors
@Typed(int)
class Integer(Descriptor):
    pass

@Unsigned
class UnsignedInteger(Integer):
    pass

@Typed(float)
class Float(Descriptor):
    pass

@Unsigned
class UnsignedFloat(Float):
    pass

@Typed(str)
class String(Descriptor):
    pass

@MaxSized
class SizedString(String):
    pass

```

这种方式定义的类跟之前的效果一样，而且执行速度会更快。设置一个简单的类型属性的值，装饰器方式要比之前的混入类的方式几乎快100%。现在你应该庆幸自己读完了本节全部内容了吧？ ^_^

8.14 实现自定义容器

问题

你想实现一个自定义的类来模拟内置的容器类功能，比如列表和字典。但是你不确定到底要实现哪些方法。

解决方案

`collections` 定义了很多抽象基类，当你想自定义容器类的时候它们会非常有用。比如你想让你的类支持迭代，那就让你的类继承 `collections.Iterable` 即可：

```
import collections
class A(collections.Iterable):
    pass
```

不过你需要实现 `collections.Iterable` 所有的抽象方法，否则会报错：

```
>>> a = A()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class A with abstract methods __iter__
>>>
```

你只要实现 `__iter__()` 方法就不会报错了(参考4.2和4.7小节)。

你可以先试着去实例化一个对象，在错误提示中可以找到需要实现哪些方法：

```
>>> import collections
>>> collections.Sequence()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with abstract methods \
__getitem__, __len__
>>>
```

下面是一个简单的示例，继承自上面`Sequence`抽象类，并且实现元素按照顺序存储：

```

class SortedItems(collections.Sequence):
    def __init__(self, initial=None):
        self._items = sorted(initial) if initial is not None else []

    # Required sequence methods
    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    # Method for adding an item in the right location
    def add(self, item):
        bisect.insort(self._items, item)

items = SortedItems([5, 1, 3])
print(list(items))
print(items[0], items[-1])
items.add(2)
print(list(items))

```

可以看到，`SortedItems`跟普通的序列没什么两样，支持所有常用操作，包括索引、迭代、包含判断，甚至是切片操作。

这里面使用到了 `bisect` 模块，它是一个在排序列表中插入元素的高效方式。可以保证元素插入后还保持顺序。

讨论

使用 `collections` 中的抽象基类可以确保你自定义的容器实现了所有必要的方法。并且还能简化类型检查。你的自定义容器会满足大部分类型检查需要，如下所示：

```

>>> items = SortedItems()
>>> import collections
>>> isinstance(items, collections.Iterable)
True
>>> isinstance(items, collections.Sequence)
True
>>> isinstance(items, collections.Container)
True
>>> isinstance(items, collections.Sized)
True
>>> isinstance(items, collections.Mapping)
False
>>>

```


`collections` 中很多抽象类会为一些常见容器操作提供默认的实现，这样一来你只需要实现那些你最感兴趣的方法即可。假设你的类继承自 `collections.MutableSequence`，如下：

```
class Items(collections.MutableSequence):
    def __init__(self, initial=None):
        self._items = list(initial) if initial is not None else []

    # Required sequence methods
    def __getitem__(self, index):
        print('Getting:', index)
        return self._items[index]

    def __setitem__(self, index, value):
        print('Setting:', index, value)
        self._items[index] = value

    def __delitem__(self, index):
        print('Deleting:', index)
        del self._items[index]

    def insert(self, index, value):
        print('Inserting:', index, value)
        self._items.insert(index, value)

    def __len__(self):
        print('Len')
        return len(self._items)
```

如果你创建 `Items` 的实例，你会发现它支持几乎所有的核心列表方法(如`append()`、`remove()`、`count()`等)。下面是使用演示：

```
>>> a = Items([1, 2, 3])
>>> len(a)
Len
3
>>> a.append(4)
Len
Inserting: 3 4
>>> a.append(2)
Len
Inserting: 4 2
>>> a.count(2)
Getting: 0
Getting: 1
Getting: 2
Getting: 3
Getting: 4
Getting: 5
2
>>> a.remove(3)
Getting: 0
Getting: 1
Getting: 2
Deleting: 2
>>>
```

本小节只是对Python抽象类功能的抛砖引玉。`numbers` 模块提供了一个类似的跟整数类型相关的抽象类型集合。可以参考8.12小节来构造更多自定义抽象基类。

8.15 属性的代理访问

问题

你想将某个实例的属性访问代理到内部另一个实例中去，目的可能是作为继承的一个替代方法或者实现代理模式。

解决方案

简单来说，代理是一种编程模式，它将某个操作转移给另外一个对象来实现。最简单的形式可能是像下面这样：

```

class A:
    def spam(self, x):
        pass

    def foo(self):
        pass

class B1:
    """简单的代理"""

    def __init__(self):
        self._a = A()

    def spam(self, x):
        # Delegate to the internal self._a instance
        return self._a.spam(x)

    def foo(self):
        # Delegate to the internal self._a instance
        return self._a.foo()

    def bar(self):
        pass

```

如果仅仅就两个方法需要代理，那么像这样写就足够了。但是，如果有大量的方法需要代理，那么使用 `__getattr__()` 方法或许或更好些：

```

class B2:
    """使用__getattr__的代理，代理方法比较多时候"""

    def __init__(self):
        self._a = A()

    def bar(self):
        pass

    # Expose all of the methods defined on class A
    def __getattr__(self, name):
        """这个方法在访问的attribute不存在的时候被调用
        the __getattr__() method is actually a fallback method
        that only gets called when an attribute is not found"""
        return getattr(self._a, name)

```

`__getattr__` 方法是在访问attribute不存在的时候被调用，使用演示：

```

b = B()
b.bar() # Calls B.bar() (exists on B)
b.spam(42) # Calls B.__getattr__('spam') and delegates to A.spam

```

另外一个代理例子是实现代理模式，例如：

```
# A proxy class that wraps around another object, but
# exposes its public attributes
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        print('getattr:', name)
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
        else:
            print('setattr:', name, value)
            setattr(self._obj, name, value)

    # Delegate attribute deletion
    def __delattr__(self, name):
        if name.startswith('_'):
            super().__delattr__(name)
        else:
            print('delattr:', name)
            delattr(self._obj, name)
```

使用这个代理类时，你只需要用它来包装下其他类即可：

```
class Spam:
    def __init__(self, x):
        self.x = x

    def bar(self, y):
        print('Spam.bar:', self.x, y)

# Create an instance
s = Spam(2)
# Create a proxy around it
p = Proxy(s)
# Access the proxy
print(p.x) # Outputs 2
p.bar(3) # Outputs "Spam.bar: 2 3"
p.x = 37 # Changes s.x to 37
```

通过自定义属性访问方法，你可以用不同方式自定义代理类行为(比如加入日志功能、只读访问等)。

讨论

代理类有时候可以作为继承的替代方案。例如，一个简单的继承如下：

```
class A:
    def spam(self, x):
        print('A.spam', x)
    def foo(self):
        print('A.foo')

class B(A):
    def spam(self, x):
        print('B.spam')
        super().spam(x)
    def bar(self):
        print('B.bar')
```

使用代理的话，就是下面这样：

```
class A:
    def spam(self, x):
        print('A.spam', x)
    def foo(self):
        print('A.foo')

class B:
    def __init__(self):
        self._a = A()
    def spam(self, x):
        print('B.spam', x)
        self._a.spam(x)
    def bar(self):
        print('B.bar')
    def __getattr__(self, name):
        return getattr(self._a, name)
```

当实现代理模式时，还有些细节需要注意。首先，`__getattr__()` 实际是一个后备方法，只有在属性不存在时才会调用。因此，如果代理类实例本身有这个属性的话，那么不会触发这个方法的。另外，`__setattr__()` 和 `__delattr__()` 需要额外的魔法来区分代理实例和被代理实例 `_obj` 的属性。一个通常的约定是只代理那些不以下划线 `_` 开头的属性(代理类只暴露被代理类的公共属性)。

还有一点需要注意的是，`__getattr__()` 对于大部分以双下划线(`__`)开始和结尾的属性并不适用。比如，考虑如下的类：

```
class ListLike:
    """__getattr__对于双下划线开始和结尾的方法是不能用的，需要一个个去重定义"""

    def __init__(self):
        self._items = []

    def __getattr__(self, name):
        return getattr(self._items, name)
```

如果是创建一个ListLike对象，会发现它支持普通的列表方法，如`append()`和`insert()`，但是却不支持`len()`、元素查找等。例如：

```
>>> a = ListLike()
>>> a.append(2)
>>> a.insert(0, 1)
>>> a.sort()
>>> len(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'ListLike' has no len()
>>> a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'ListLike' object does not support indexing
>>>
```

为了让它支持这些方法，你必须手动的实现这些方法代理：

```
class ListLike:
    """__getattr__对于双下划线开始和结尾的方法是不能用的，需要一个个去重定义"""

    def __init__(self):
        self._items = []

    def __getattr__(self, name):
        return getattr(self._items, name)

    # Added special methods to support certain list operations
    def __len__(self):
        return len(self._items)

    def __getitem__(self, index):
        return self._items[index]

    def __setitem__(self, index, value):
        self._items[index] = value

    def __delitem__(self, index):
        del self._items[index]
```

11.8小节还有一个在远程方法调用环境中使用代理的例子。

8.16 在类中定义多个构造器

问题

你想实现一个类，除了使用 `__init__()` 方法外，还有其他方式可以初始化它。

解决方案

为了实现多个构造器，你需要使用到类方法。例如：

```
import time

class Date:
    """方法一：使用类方法"""
    # Primary constructor
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    # Alternate constructor
    @classmethod
    def today(cls):
        t = time.localtime()
        return cls(t.tm_year, t.tm_mon, t.tm_mday)
```

直接调用类方法即可，下面是使用示例：

```
a = Date(2012, 12, 21) # Primary
b = Date.today() # Alternate
```

讨论

类方法的一个主要用途就是定义多个构造器。它接受一个 `class` 作为第一个参数(`cls`)。你应该注意到了这个类被用来创建并返回最终的实例。在继承时也能工作的很好：

```
class NewDate(Date):
    pass

c = Date.today() # Creates an instance of Date (cls=Date)
d = NewDate.today() # Creates an instance of NewDate (cls=NewDate)
```

8.17 创建不调用 `__init__` 方法的实例

问题

你想创建一个实例，但是希望绕过执行 `__init__()` 方法。

解决方案

可以通过 `__new__()` 方法创建一个未初始化的实例。例如考虑如下这个类：

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

下面演示如何不调用 `__init__()` 方法来创建这个Date实例：

```
>>> d = Date.__new__(Date)
>>> d
<__main__.Date object at 0x1006716d0>
>>> d.year
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Date' object has no attribute 'year'
>>>
```

结果可以看到，这个Date实例的属性year还不存在，所以你需要手动初始化：

```
>>> data = {'year':2012, 'month':8, 'day':29}
>>> for key, value in data.items():
...     setattr(d, key, value)
...
>>> d.year
2012
>>> d.month
8
>>>
```

讨论

当我们在反序列对象或者实现某个类方法构造函数时需要绕过 `__init__()` 方法来创建对象。例如，对于上面的Date来讲，有时候你可能会像下面这样定义一个新的构造函数

```
today() :
```

```
from time import localtime

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        d = cls.__new__(cls)
        t = localtime()
        d.year = t.tm_year
        d.month = t.tm_mon
        d.day = t.tm_mday
        return d
```

同样，在你反序列化JSON数据时产生一个如下的字典对象：

```
data = { 'year': 2012, 'month': 8, 'day': 29 }
```

如果你想将它转换成一个Date类型实例，可以使用上面的技术。

当你通过这种非常规方式来创建实例的时候，最好不要直接去访问底层实例字典，除非你真的清楚所有细节。否则的话，如果这个类使用了 `__slots__`、`properties`、`descriptors` 或其他高级技术的时候代码就会失效。而这时候使用 `setattr()` 方法会让你的代码变得更加通用。

8.18 利用Mixins扩展类功能

问题

你有很多有用的方法，想使用它们来扩展其他类的功能。但是这些类并没有任何继承的关系。因此你不能简单的将这些方法放入一个基类，然后被其他类继承。

解决方案

通常当你想自定义类的时候会碰上这些问题。可能是某个库提供了一些基础类，你可以利用它们来构造你自己的类。

假设你想扩展映射对象，给它们添加日志、唯一性设置、类型检查等等功能。下面是一些混入类：

```

class LoggedMappingMixin:
    """
    Add logging to get/set/delete operations for debugging.
    """
    __slots__ = () # 混入类都没有实例变量，因为直接实例化混入类没有任何意义

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return super().__getitem__(key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return super().__setitem__(key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return super().__delitem__(key)

class SetOnceMappingMixin:
    """
    Only allow a key to be set once.
    """
    __slots__ = ()

    def __setitem__(self, key, value):
        if key in self:
            raise KeyError(str(key) + ' already set')
        return super().__setitem__(key, value)

class StringKeysMappingMixin:
    """
    Restrict keys to strings only
    """
    __slots__ = ()

    def __setitem__(self, key, value):
        if not isinstance(key, str):
            raise TypeError('keys must be strings')
        return super().__setitem__(key, value)

```

这些类单独使用起来没有任何意义，事实上如果你去实例化任何一个类，除了产生异常外没有任何作用。它们是用来通过多继承来和其他映射对象混入使用的。例如：

```

class LoggedDict(LoggedMappingMixin, dict):
    pass

d = LoggedDict()
d['x'] = 23
print(d['x'])
del d['x']

from collections import defaultdict

class SetOnceDefaultDict(SetOnceMappingMixin, defaultdict):
    pass

d = SetOnceDefaultDict(list)
d['x'].append(2)
d['x'].append(3)
# d['x'] = 23 # KeyError: 'x already set'

```

这个例子中，可以看到混入类跟其他已存在的类(比如dict、defaultdict和OrderedDict)结合起来使用，一个接一个。结合后就能发挥正常功效了。

讨论

混入类在标志库中很多地方都出现过，通常都是用来像上面那样扩展某些类的功能。它们也是多继承的一个主要用途。比如，当你编写网络代码时候，你会经常使用 `socketserver` 模块中的 `ThreadingMixIn` 来给其他网络相关类增加多线程支持。例如，下面是一个多线程的XML-RPC服务：

```

from xmlrpc.server import SimpleXMLRPCServer
from socketserver import ThreadingMixIn
class ThreadedXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
    pass

```

同时在一些大型库和框架中也会发现混入类的使用，用途同样是增强已存在的类的功能和一些可选特征。

对于混入类，有几点需要记住。首先是，混入类不能被实例化使用。其次，混入类没有自己的状态信息，也就是说它们并没有定义 `__init__()` 方法，并且没有实例属性。这也是为什么我们在上面明确定义了 `__slots__ = ()`。

还有一种实现混入类的方式就是使用类装饰器，如下所示：

```

def LoggedMapping(cls):
    """第二种方式：使用类装饰器"""
    cls_getitem = cls.__getitem__
    cls_setitem = cls.__setitem__
    cls_delitem = cls.__delitem__

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return cls_getitem(self, key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return cls_setitem(self, key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return cls_delitem(self, key)

    cls.__getitem__ = __getitem__
    cls.__setitem__ = __setitem__
    cls.__delitem__ = __delitem__
    return cls

@LoggedMapping
class LoggedDict(dict):
    pass

```

这个效果跟之前的是一样的，而且不再需要使用多继承了。参考9.12小节获取更多类装饰器的信息，参考8.13小节查看更多混入类和类装饰器的例子。

8.19 实现状态对象或者状态机

问题

你想实现一个状态机或者是在不同状态下执行操作的对象，但是又不想在代码中出现太多的条件判断语句。

解决方案

在很多程序中，有些对象会根据状态的不同来执行不同的操作。比如考虑如下的一个连接对象：

```

class Connection:
    """普通方案，好多个判断语句，效率低下~~"""

    def __init__(self):
        self.state = 'CLOSED'

    def read(self):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('reading')

    def write(self, data):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('writing')

    def open(self):
        if self.state == 'OPEN':
            raise RuntimeError('Already open')
        self.state = 'OPEN'

    def close(self):
        if self.state == 'CLOSED':
            raise RuntimeError('Already closed')
        self.state = 'CLOSED'

```

这样写有很多缺点，首先是代码太复杂了，好多的条件判断。其次是执行效率变低，因为一些常见的操作比如read()、write()每次执行前都需要执行检查。

一个更好的办法是为每个状态定义一个对象：

```

class Connection1:
    """新方案——对每个状态定义一个类"""

    def __init__(self):
        self.new_state(ClosedConnectionState)

    def new_state(self, newstate):
        self._state = newstate
        # Delegate to the state class

    def read(self):
        return self._state.read(self)

    def write(self, data):
        return self._state.write(self, data)

    def open(self):
        return self._state.open(self)

    def close(self):
        return self._state.close(self)

```

```

# Connection state base class
class ConnectionState:
    @staticmethod
    def read(conn):
        raise NotImplementedError()

    @staticmethod
    def write(conn, data):
        raise NotImplementedError()

    @staticmethod
    def open(conn):
        raise NotImplementedError()

    @staticmethod
    def close(conn):
        raise NotImplementedError()

# Implementation of different states
class ClosedConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        raise RuntimeError('Not open')

    @staticmethod
    def write(conn, data):
        raise RuntimeError('Not open')

    @staticmethod
    def open(conn):
        conn.new_state(OpenConnectionState)

    @staticmethod
    def close(conn):
        raise RuntimeError('Already closed')

class OpenConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        print('reading')

    @staticmethod
    def write(conn, data):
        print('writing')

    @staticmethod
    def open(conn):
        raise RuntimeError('Already open')

    @staticmethod
    def close(conn):
        conn.new_state(ClosedConnectionState)

```

下面是使用演示：

```
>>> c = Connection()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 10, in read
    return self._state.read(self)
  File "example.py", line 43, in read
    raise RuntimeError('Not open')
RuntimeError: Not open
>>> c.open()
>>> c._state
<class '__main__.OpenConnectionState'>
>>> c.read()
reading
>>> c.write('hello')
writing
>>> c.close()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>>
```

讨论

如果代码中出现太多的条件判断语句的话，代码就会变得难以维护和阅读。这里的解决方案是将每个状态抽取出来定义成一个类。

这里看上去有点奇怪，每个状态对象都只有静态方法，并没有存储任何的实例属性数据。实际上，所有状态信息都只存储在 `Connection` 实例中。在基类中定义的 `NotImplementedError` 是为了确保子类实现了相应的方法。这里你或许还想使用8.12小节讲解的抽象基类方式。

设计模式中有一种模式叫状态模式，这一小节算是一个初步入门！

8.20 通过字符串调用对象方法

问题

你有一个字符串形式的方法名称，想通过它调用某个对象的对应方法。

解决方案

最简单的情况，可以使用 `getattr()`：

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point({!r},{!r})'.format(self.x, self.y)

    def distance(self, x, y):
        return math.hypot(self.x - x, self.y - y)

p = Point(2, 3)
d = getattr(p, 'distance')(0, 0) # Calls p.distance(0, 0)
```

另外一种方法是使用 `operator.methodcaller()`，例如：

```
import operator
operator.methodcaller('distance', 0, 0)(p)
```

当你需要通过相同的参数多次调用某个方法时，使用 `operator.methodcaller` 就很方便了。比如你需要排序一系列的点，就可以这样做：

```
points = [
    Point(1, 2),
    Point(3, 0),
    Point(10, -3),
    Point(-5, -7),
    Point(-1, 8),
    Point(3, 2)
]
# Sort by distance from origin (0, 0)
points.sort(key=operator.methodcaller('distance', 0, 0))
```

讨论

调用一个方法实际上是两部独立操作，第一步是查找属性，第二步是函数调用。因此，为了调用某个方法，你可以首先通过 `getattr()` 来查找到这个属性，然后再去以函数方式调用它即可。

`operator.methodcaller()` 创建一个可调用对象，并同时提供所有必要参数，然后调用的时候只需要将实例对象传递给它即可，比如：

```
>>> p = Point(3, 4)
>>> d = operator.methodcaller('distance', 0, 0)
>>> d(p)
5.0
>>>
```

通过方法名称字符串来调用方法通常出现在需要模拟 `case` 语句或实现访问者模式的时候。参考下一小节获取更多高级例子。

8.21 实现访问者模式

问题

你要处理由大量不同类型的对象组成的复杂数据结构，每一个对象都需要需要进行不同的处理。比如，遍历一个树形结构，然后根据每个节点的相应状态执行不同的操作。

解决方案

这里遇到的问题在编程领域中是很普遍的，有时候会构建一个由大量不同对象组成的数据结构。假设你要写一个表示数学表达式的程序，那么你可能需要定义如下的类：

```

class Node:
    pass

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

```

然后利用这些类构建嵌套数据结构，如下所示：

```

# Representation of 1 + 2 * (3 - 4) / 5
t1 = Sub(Number(3), Number(4))
t2 = Mul(Number(2), t1)
t3 = Div(t2, Number(5))
t4 = Add(Number(1), t3)

```

这样做的问题是由于每个表达式，每次都要重新定义一遍，有没有一种更通用的方式让它支持所有的数字和操作符呢。这里我们使用访问者模式可以达到这样的目的：

```

class NodeVisitor:
    def visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))

```

为了使用这个类，可以定义一个类继承它并且实现各种 `visit_Name()` 方法，其中Name是node类型。例如，如果你想求表达式的值，可以这样写：

```

class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -node.operand

```

使用示例：

```

>>> e = Evaluator()
>>> e.visit(t4)
0.6
>>>

```

作为一个不同的例子，下面定义一个类在一个栈上面将一个表达式转换成多个操作序列：

```

class StackCode(NodeVisitor):
    def generate_code(self, node):
        self.instructions = []
        self.visit(node)
        return self.instructions

    def visit_Number(self, node):
        self.instructions.append(('PUSH', node.value))

    def binop(self, node, instruction):
        self.visit(node.left)
        self.visit(node.right)
        self.instructions.append((instruction,))

    def visit_Add(self, node):
        self.binop(node, 'ADD')

    def visit_Sub(self, node):
        self.binop(node, 'SUB')

    def visit_Mul(self, node):
        self.binop(node, 'MUL')

    def visit_Div(self, node):
        self.binop(node, 'DIV')

    def unaryop(self, node, instruction):
        self.visit(node.operand)
        self.instructions.append((instruction,))

    def visit_Negate(self, node):
        self.unaryop(node, 'NEG')

```

使用示例:

```

>>> s = StackCode()
>>> s.generate_code(t4)
[('PUSH', 1), ('PUSH', 2), ('PUSH', 3), ('PUSH', 4), ('SUB',),
 ('MUL',), ('PUSH', 5), ('DIV',), ('ADD',)]
>>>

```

讨论

刚开始的时候你可能会写大量的if/else语句来实现，这里访问者模式的好处就是通过 `getattr()` 来获取相应的方法，并利用递归来遍历所有的节点：

```
def binop(self, node, instruction):
    self.visit(node.left)
    self.visit(node.right)
    self.instructions.append((instruction,))
```

还有一点需要指出的是，这种技术也是实现其他语言中switch或case语句的方式。比如，如果你正在写一个HTTP框架，你可能会写这样一个请求分发的控制器：

```
class HTTPHandler:
    def handle(self, request):
        methname = 'do_' + request.request_method
        getattr(self, methname)(request)
    def do_GET(self, request):
        pass
    def do_POST(self, request):
        pass
    def do_HEAD(self, request):
        pass
```

访问者模式一个缺点就是它严重依赖递归，如果数据结构嵌套层次太深可能会有问题，有时候会超过Python的递归深度限制(参考 `sys.getrecursionlimit()`)。

可以参照8.22小节，利用生成器或迭代器来实现非递归遍历算法。

在跟解析和编译相关的编程中使用访问者模式是非常常见的。Python本身的 `ast` 模块值的关注下，可以去看看源码。9.24小节演示了一个利用 `ast` 模块来处理Python源代码的例子。

8.22 不用递归实现访问者模式

问题

你使用访问者模式遍历一个很深的嵌套树形数据结构，并且因为超过嵌套层级限制而失败。你想消除递归，并同时保持访问者编程模式。

解决方案

通过巧妙的使用生成器可以在树遍历或搜索算法中消除递归。在8.21小节中，我们给出了一个访问者类。下面我们利用一个栈和生成器重新实现这个类：

```

import types

class Node:
    pass

class NodeVisitor:
    def visit(self, node):
        stack = [node]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Node):
                    stack.append(self._visit(stack.pop()))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()

        return last_result

    def _visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))

```

如果你使用这个类，也能达到相同的效果。事实上你完全可以将它作为上一节中的访问者模式的替代实现。考虑如下代码，遍历一个表达式的树：

```

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):

```

```

    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

# A sample visitor class that evaluates expressions
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -self.visit(node.operand)

if __name__ == '__main__':
    # 1 + 2*(3-4) / 5
    t1 = Sub(Number(3), Number(4))
    t2 = Mul(Number(2), t1)
    t3 = Div(t2, Number(5))
    t4 = Add(Number(1), t3)
    # Evaluate it
    e = Evaluator()
    print(e.visit(t4)) # Outputs 0.6

```

如果嵌套层次太深那么上述的Evaluator就会失效:


```

>>> a = Number(0)
>>> for n in range(1, 100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
Traceback (most recent call last):
...
  File "visitor.py", line 29, in _visit
return meth(node)
  File "visitor.py", line 67, in visit_Add
return self.visit(node.left) + self.visit(node.right)
RuntimeError: maximum recursion depth exceeded
>>>

```

现在我们稍微修改下上面的Evaluator:

```

class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        yield (yield node.left) + (yield node.right)

    def visit_Sub(self, node):
        yield (yield node.left) - (yield node.right)

    def visit_Mul(self, node):
        yield (yield node.left) * (yield node.right)

    def visit_Div(self, node):
        yield (yield node.left) / (yield node.right)

    def visit_Negate(self, node):
        yield - (yield node.operand)

```

再次运行，就不会报错了:

```

>>> a = Number(0)
>>> for n in range(1,100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
4999950000
>>>

```

如果你还想添加其他自定义逻辑也没问题：

```
class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        print('Add:', node)
        lhs = yield node.left
        print('left=', lhs)
        rhs = yield node.right
        print('right=', rhs)
        yield lhs + rhs
    ...
```

下面是简单的测试：

```
>>> e = Evaluator()
>>> e.visit(t4)
Add: <__main__.Add object at 0x1006a8d90>
left= 1
right= -0.4
0.6
>>>
```

讨论

这一小节我们演示了生成器和协程在程序控制流方面的强大功能。避免递归的一个通常方法是使用一个栈或队列的数据结构。例如，深度优先的遍历算法，第一次碰到一个节点时将其压入栈中，处理完后弹出栈。`visit()`方法的核心思路就是这样。

另外一个需要理解的就是生成器中`yield`语句。当碰到`yield`语句时，生成器会返回一个数据并暂时挂起。上面的例子使用这个技术来代替了递归。例如，之前我们是这样写递归：

```
value = self.visit(node.left)
```

现在换成`yield`语句：

```
value = yield node.left
```

它会将 `node.left` 返回给 `visti()` 方法，然后 `visti()` 方法调用那个节点相应的 `vist_Name()` 方法。 `yield` 暂时将程序控制器让出给调用者，当执行完后，结果会赋值给 `value`，

看完这一小节，你也许想去寻找其它没有 `yield` 语句的方案。但是这么做没有必要，你必须处理很多棘手的问题。例如，为了消除递归，你必须维护一个栈结构，如果不使用生成器，代码会变得很臃肿，到处都是栈操作语句、回调函数等。实际上，使用 `yield` 语句可以让你写出非常漂亮的代码，它消除了递归但是看上去又很像递归实现，代码很简洁。

8.23 循环引用数据结构的内存管理

问题

你的程序创建了很多循环引用数据结构(比如树、图、观察者模式等)，你碰到了内存管理难题。

解决方案

一个简单的循环引用数据结构例子就是一个树形结构，双亲节点有指针指向孩子节点，孩子节点又返回来指向双亲节点。这种情况下，可以考虑使用 `weakref` 库中的弱引用。例如：

```
import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self._parent = None
        self.children = []

    def __repr__(self):
        return 'Node({!r:})'.format(self.value)

    # property that manages the parent as a weak-reference
    @property
    def parent(self):
        return None if self._parent is None else self._parent()

    @parent.setter
    def parent(self, node):
        self._parent = weakref.ref(node)

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

这种是想方式允许parent静默终止。例如：

```
>>> root = Node('parent')
>>> c1 = Node('child')
>>> root.add_child(c1)
>>> print(c1.parent)
Node('parent')
>>> del root
>>> print(c1.parent)
None
>>>
```

讨论

循环引用的数据结构在Python中是一个很棘手的问题，因为正常的垃圾回收机制不能适用于这种情形。例如考虑如下代码：

```
# Class just to illustrate when deletion occurs
class Data:
    def __del__(self):
        print('Data.__del__')

# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

下面我们使用这个代码来做一些垃圾回收试验：

```
>>> a = Data()
>>> del a # Immediately deleted
Data.__del__
>>> a = Node()
>>> del a # Immediately deleted
Data.__del__
>>> a = Node()
>>> a.add_child(Node())
>>> del a # Not deleted (no message)
>>>
```

可以看到，最后一个的删除时打印语句没有出现。原因是Python的垃圾回收机制是基于简单的引用计数。当一个对象的引用数变成0的时候才会立即删除掉。而对于循环引用这个条件永远不会成立。因此，在上面例子中最后部分，父节点和孩子节点互相拥有对方的引用，导致每个对象的引用计数都不可能变成0。

Python有另外的垃圾回收器来专门针对循环引用的，但是你永远不知道它什么时候会触发。另外你还可以手动的触发它，但是代码看上去很挫：

```
>>> import gc
>>> gc.collect() # Force collection
Data.__del__
Data.__del__
>>>
```

如果循环引用的对象自己还定义了自己的 `__del__()` 方法，那么会让情况变得更糟糕。假设你像下面这样给Node定义自己的 `__del__()` 方法：

```
# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        child.parent = self

    # NEVER DEFINE LIKE THIS.
    # Only here to illustrate pathological behavior
    def __del__(self):
        del self.data
        del self.parent
        del self.children
```

这种情况下，垃圾回收永远都不会去回收这个对象的，还会导致内存泄露。如果你试着去运行它会发现， `Data.__del__` 消息永远不会出现了，甚至在你强制内存回收时：

```
>>> a = Node()
>>> a.add_child(Node())
>>> del a # No message (not collected)
>>> import gc
>>> gc.collect() # No message (not collected)
>>>
```

弱引用消除了引用循环的这个问题，本质来讲，弱引用就是一个对象指针，它不会增加它的引用计数。你可以通过 `weakref` 来创建弱引用。例如：

```
>>> import weakref
>>> a = Node()
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x100581f70; to 'Node' at 0x1005c5410>
>>>
```

为了访问弱引用所引用的对象，你可以像函数一样去调用它即可。如果那个对象还存在就会返回它，否则就返回一个 `None`。由于原始对象的引用计数没有增加，那么就可以去删除它了。例如：

```
>>> print(a_ref())
<__main__.Node object at 0x1005c5410>
>>> del a
Data.__del__
>>> print(a_ref())
None
>>>
```

通过这里演示的弱引用技术，你会发现不再有循环引用问题了，一旦某个节点不被使用了，垃圾回收器立即回收它。你还能参考8.25小节关于弱引用的另外一个例子。

8.24 让类支持比较操作

问题

你想让某个类的实例支持标准的比较运算(比如 `>=`, `!=`, `<=`, `<` 等)，但是又不想去实现那一大丢的特殊方法。

解决方案

Python类对每个比较操作都需要实现一个特殊方法来支持。例如为了支持`>=`操作符，你需要定义一个 `__ge__()` 方法。尽管定义一个方法没什么问题，但如果要你实现所有可能的比较方法那就有点烦人了。

装饰器 `functools.total_ordering` 就是用来简化这个处理的。使用它来装饰一个类，你只需定义一个 `__eq__()` 方法，外加其他方法(`__lt__`, `__le__`, `__gt__`, or `__ge__`)中的一个即可。然后装饰器会自动为你填充其它比较方法。

作为例子，我们构建一些房子，然后给它们增加一些房间，最后通过房子大小来比较它们：

```
from functools import total_ordering

class Room:
    def __init__(self, name, length, width):
        self.name = name
        self.length = length
        self.width = width
        self.square_feet = self.length * self.width

@total_ordering
class House:
    def __init__(self, name, style):
        self.name = name
        self.style = style
        self.rooms = list()

    @property
    def living_space_footage(self):
        return sum(r.square_feet for r in self.rooms)

    def add_room(self, room):
        self.rooms.append(room)

    def __str__(self):
        return '{}: {} square foot {}'.format(self.name,
            self.living_space_footage,
            self.style)

    def __eq__(self, other):
        return self.living_space_footage == other.living_space_footage

    def __lt__(self, other):
        return self.living_space_footage < other.living_space_footage
```

这里我们只是给House类定义了两个方法：`__eq__()` 和 `__lt__()`，它就能支持所有的比较操作：

```

# Build a few houses, and add rooms to them
h1 = House('h1', 'Cape')
h1.add_room(Room('Master Bedroom', 14, 21))
h1.add_room(Room('Living Room', 18, 20))
h1.add_room(Room('Kitchen', 12, 16))
h1.add_room(Room('Office', 12, 12))
h2 = House('h2', 'Ranch')
h2.add_room(Room('Master Bedroom', 14, 21))
h2.add_room(Room('Living Room', 18, 20))
h2.add_room(Room('Kitchen', 12, 16))
h3 = House('h3', 'Split')
h3.add_room(Room('Master Bedroom', 14, 21))
h3.add_room(Room('Living Room', 18, 20))
h3.add_room(Room('Office', 12, 16))
h3.add_room(Room('Kitchen', 15, 17))
houses = [h1, h2, h3]
print('Is h1 bigger than h2?', h1 > h2) # prints True
print('Is h2 smaller than h3?', h2 < h3) # prints True
print('Is h2 greater than or equal to h1?', h2 >= h1) # Prints False
print('Which one is biggest?', max(houses)) # Prints 'h3: 1101-square-foot Split'
print('Which is smallest?', min(houses)) # Prints 'h2: 846-square-foot Ranch'

```

讨论

其实 `total_ordering` 装饰器也没那么神秘。它就是定义了一个从每个比较支持方法到所有需要定义的其他方法的一个映射而已。比如你定义了 `__le__()` 方法，那么它就被用来构建所有其他的需要定义的那些特殊方法。实际上就是在类里面像下面这样定义了一些特殊方法：

```

class House:
    def __eq__(self, other):
        pass
    def __lt__(self, other):
        pass
    # Methods created by @total_ordering
    __le__ = lambda self, other: self < other or self == other
    __gt__ = lambda self, other: not (self < other or self == other)
    __ge__ = lambda self, other: not (self < other)
    __ne__ = lambda self, other: not self == other

```

当然，你自己去写也很容易，但是使用 `@total_ordering` 可以简化代码，何乐而不为呢。

8.25 创建缓存实例

问题

在创建一个类的对象时，如果之前使用同样参数创建过这个对象，你想返回它的缓存引用。

解决方案

这种通常是因为你希望相同参数创建的对象是单例的。在很多库中都有实际的例子，比如 `logging` 模块，使用相同的名称创建的 `logger` 实例永远只有一个。例如：

```
>>> import logging
>>> a = logging.getLogger('foo')
>>> b = logging.getLogger('bar')
>>> a is b
False
>>> c = logging.getLogger('foo')
>>> a is c
True
>>>
```

为了达到这样的效果，你需要使用一个和类本身分开的工厂函数，例如：

```
# The class in question
class Spam:
    def __init__(self, name):
        self.name = name

# Caching support
import weakref
_spam_cache = weakref.WeakValueDictionary()
def get_spam(name):
    if name not in _spam_cache:
        s = Spam(name)
        _spam_cache[name] = s
    else:
        s = _spam_cache[name]
    return s
```

然后做一个测试，你会发现跟之前那个日志对象的创建行为是一致的：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> a is b
False
>>> c = get_spam('foo')
>>> a is c
True
>>>
```

讨论

编写一个工厂函数来修改普通的实例创建行为通常是一个比较简单的方法。但是我们还能否找到更优雅的方案呢？

例如，你可能会考虑重新定义类的 `__new__()` 方法，就像下面这样：

```
# Note: This code doesn't quite work
import weakref

class Spam:
    _spam_cache = weakref.WeakValueDictionary()
    def __new__(cls, name):
        if name in cls._spam_cache:
            return cls._spam_cache[name]
        else:
            self = super().__new__(cls)
            cls._spam_cache[name] = self
            return self
    def __init__(self, name):
        print('Initializing Spam')
        self.name = name
```

初看起来好像可以达到预期效果，但是问题是 `__init__()` 每次都会被调用，不管这个实例是否被缓存了。例如：

```
>>> s = Spam('Dave')
Initializing Spam
>>> t = Spam('Dave')
Initializing Spam
>>> s is t
True
>>>
```

这个或许不是你想要的效果，因此这种方法并不可取。

上面我们使用到了弱引用计数，对于垃圾回收来讲是很有帮助的，关于这个我们在8.23小节已经讲过了。当我们保持实例缓存时，你可能只想在程序中使用到它们时才保存。一个 `WeakValueDictionary` 实例只会保存那些在其它地方还在被使用的实例。否则的话，只要实例不再被使用了，它就从字典中被移除了。观察下下面的测试结果：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> c = get_spam('foo')
>>> list(_spam_cache)
['foo', 'bar']
>>> del a
>>> del c
>>> list(_spam_cache)
['bar']
>>> del b
>>> list(_spam_cache)
[]
>>>
```

对于大部分程序而已，这里代码已经够用了。不过还是有一些更高级的实现值得了解下。

首先是这里使用到了一个全局变量，并且工厂函数跟类放在一块。我们可以通过将缓存代码放到一个单独的缓存管理器中：

```

import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()

    def get_spam(self, name):
        if name not in self._cache:
            s = Spam(name)
            self._cache[name] = s
        else:
            s = self._cache[name]
        return s

    def clear(self):
        self._cache.clear()

class Spam:
    manager = CachedSpamManager()
    def __init__(self, name):
        self.name = name

    def get_spam(name):
        return Spam.manager.get_spam(name)

```

这样的话代码更清晰，并且也更灵活，我们可以增加更多的缓存管理机制，只需要替代 `manager` 即可。

还有一点就是，我们暴露了类的实例化给用户，用户很容易去直接实例化这个类，而不是使用工厂方法，如：

```

>>> a = Spam('foo')
>>> b = Spam('foo')
>>> a is b
False
>>>

```

有几种方式可以防止用户这样做，第一个是将类的名字修改为以下划线(`_`)开头，提示用户别直接调用它。第二种就是让这个类的 `__init__()` 方法抛出一个异常，让它不能被初始化：

```

class Spam:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name

```

然后修改缓存管理器代码，使用 `Spam._new()` 来创建实例，而不是直接调用 `Spam()` 构造函数：

```

# -----最后的修正方案-----
class CachedSpamManager2:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()

    def get_spam(self, name):
        if name not in self._cache:
            temp = Spam3._new(name) # Modified creation
            self._cache[name] = temp
        else:
            temp = self._cache[name]
        return temp

    def clear(self):
        self._cache.clear()

class Spam3:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
        return self

```

最后这样的方案就已经足够好了。缓存和其他构造模式还可以使用9.13小节中的元类实现的更优雅一点(使用了更高级的技术)。

第九章：元编程

软件开发领域中最经典的口头禅就是“don't repeat yourself”。也就是说，任何时候当你的程序中存在高度重复(或者是通过剪切复制)的代码时，都应该想想是否有更好的解决方案。在Python当中，通常都可以通过元编程来解决这类问题。简而言之，元编程就是关

于创建操作源代码(比如修改、生成或包装原来的代码)的函数和类。主要技术是使用装饰器、类装饰器和元类。不过还有一些其他技术, 包括签名对象、使用 `exec()` 执行代码以及对内部函数和类的反射技术等。本章的主要目的是向大家介绍这些元编程技术, 并且给出实例来演示它们是怎样定制化你的源代码行为的。

Contents:

9.1 在函数上添加包装器

问题

你想在函数上添加一个包装器, 增加额外的操作处理(比如日志、计时等)。

解决方案

如果你想使用额外的代码包装一个函数, 可以定义一个装饰器函数, 例如:

```
import time
from functools import wraps

def timethis(func):
    """
    Decorator that reports the execution time.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

下面是使用装饰器的例子:

```

>>> @timethis
... def countdown(n):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown(10000000)
countdown 0.87188299392912
>>>

```

讨论

一个装饰器就是一个函数，它接受一个函数作为参数并返回一个新的函数。当你像下面这样写：

```

@timethis
def countdown(n):
    pass

```

跟像下面这样写其实效果是一样的：

```

def countdown(n):
    pass
countdown = timethis(countdown)

```

顺便说一下，内置的装饰器比如 `@staticmethod`、`@classmethod`、`@property` 原理也是一样的。例如，下面这两个代码片段是等价的：

```

class A:
    @classmethod
    def method(cls):
        pass

class B:
    # Equivalent definition of a class method
    def method(cls):
        pass
    method = classmethod(method)

```

在上面的 `wrapper()` 函数中，装饰器内部定义了一个使用 `*args` 和 `**kwargs` 来接受任意参数的函数。在这个函数里面调用了原始函数并将其结果返回，不过你还可以添加其他额外的代码(比如计时)。然后这个新的函数包装器被作为结果返回来代替原始函数。

需要强调的是装饰器并不会修改原始函数的参数签名以及返回值。使用 `*args` 和 `**kwargs` 目的就是确保任何参数都能适用。而返回结果值基本都是调用原始函数 `func(*args, **kwargs)` 的返回结果，其中 `func` 就是原始函数。

刚开始学习装饰器的时候，会使用一些简单的例子来说明，比如上面演示的这个。不过实际场景使用时，还是有一些细节问题要注意的。比如上面使用 `@wraps(func)` 注解是很重要的，它能保留原始函数的元数据(下一小节会讲到)，新手经常会忽略这个细节。接下来的几个小节我们会更加深入的讲解装饰器函数的细节问题，如果你想构造你自己的装饰器函数，需要认真看一下。

9.2 创建装饰器时保留函数元信息

问题

你写了一个装饰器作用在某个函数上，但是这个函数的重要的元信息比如名字、文档字符串、注解和参数签名都丢失了。

解决方案

任何时候你定义装饰器的时候，都应该使用 `functools` 库中的 `@wraps` 装饰器来注解底层包装函数。例如：

```
import time
from functools import wraps
def timethis(func):
    """
    Decorator that reports the execution time.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```


下面我们使用这个被包装后的函数并检查它的元信息：

```
>>> @timethis
... def countdown(n:int):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown.__name__
'countdown'
>>> countdown.__doc__
'\n\tCounts down\n\t'
>>> countdown.__annotations__
{'n': <class 'int'>}
>>>
```

讨论

在编写装饰器的时候复制元信息是一个非常重要的部分。如果你忘记了使用 `@wrap`，那么你会发现被装饰函数丢失了所有有用的信息。比如如果忽略 `@wrap` 后的效果是下面这样的：

```
>>> countdown.__name__
'wrapper'
>>> countdown.__doc__
>>> countdown.__annotations__
{}
>>>
```

`@wraps` 有一个重要特征是它能让你通过属性 `__wrapped__` 直接访问被包装函数。例如：

```
>>> countdown.__wrapped__(100000)
>>>
```

`__wrapped__` 属性还能让被装饰函数正确暴露底层的参数签名信息。例如：

```
>>> from inspect import signature
>>> print(signature(countdown))
(n:int)
>>>
```

一个很普遍的问题是怎样让装饰器去直接复制原始函数的参数签名信息，如果想自己手动实现的话需要做大量的工作，最好就简单的使用 `__wrapped__` 装饰器。通过底层的 `__wrapped__` 属性访问到函数签名信息。更多关于签名的内容可以参考9.16小节。

9.3 解除一个装饰器

问题

一个装饰器已经作用在一个函数上，你想撤销它，直接访问原始的未包装的那个函数。

解决方案

假设装饰器是通过 `@wraps` (参考9.2小节)来实现的，那么你可以通过访问 `__wrapped__` 属性来访问原始函数：

```
>>> @somedecorator
>>> def add(x, y):
...     return x + y
...
>>> orig_add = add.__wrapped__
>>> orig_add(3, 4)
7
>>>
```

讨论

直接访问未包装的原始函数在调试、内省和其他函数操作时是很有用的。但是我们这里的方案仅仅适用于在包装器中正确使用了 `@wraps` 或者直接设置了 `__wrapped__` 属性的情况。

如果有多个包装器，那么访问 `__wrapped__` 属性的行为是不可预知的，应该避免这样做。在Python3.3中，它会略过所有的包装层，比如，假如你有如下的代码：

```

from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 1')
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
def add(x, y):
    return x + y

```

下面我们在Python3.3下测试：

```

>>> add(2, 3)
Decorator 1
Decorator 2
5
>>> add.__wrapped__(2, 3)
5
>>>

```

下面我们在Python3.4下测试：

```

>>> add(2, 3)
Decorator 1
Decorator 2
5
>>> add.__wrapped__(2, 3)
Decorator 2
5
>>>

```

最后要说的是，并不是所有的装饰器都使用了 `@wraps`，因此这里的方案并不全部适用。特别的，内置的装饰器 `@staticmethod` 和 `@classmethod` 就没有遵循这个约定（它们把原始函数存储在属性 `__func__` 中）。

9.4 定义一个带参数的装饰器

问题

你想定义一个可以接受参数的装饰器

解决方案

我们用一个例子详细阐述下接受参数的处理过程。假设你想写一个装饰器，给函数添加日志功能，当时允许用户指定日志的级别和其他的选项。下面是这个装饰器的定义和使用示例：

```
from functools import wraps
import logging

def logged(level, name=None, message=None):
    """
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    """
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

初看起来，这种实现看上去很复杂，但是核心思想很简单。最外层的函数 `logged()` 接受参数并将它们作用在内部的装饰器函数上面。内层的函数 `decorate()` 接受一个函数作为参数，然后在函数上面放置一个包装器。这里的关键点是包装器是可以使用传递给 `logged()` 的参数的。

讨论

定义一个接受参数的包装器看上去比较复杂主要是因为底层的调用序列。特别的，如果你有下面这个代码：

```
@decorator(x, y, z)
def func(a, b):
    pass
```

装饰器处理过程跟下面的调用是等效的；

```
def func(a, b):
    pass
func = decorator(x, y, z)(func)
```

`decorator(x, y, z)` 的返回结果必须是一个可调用对象，它接受一个函数作为参数并包装它，可以参考9.7小节中另外一个可接受参数的包装器例子。

9.5 可自定义属性的装饰器

问题

你想写一个装饰器来包装一个函数，并且允许用户提供参数在运行时控制装饰器行为。

解决方案

引入一个访问函数，使用 `nonlocal` 来修改内部变量。然后这个访问函数被作为一个属性赋值给包装函数。

```

from functools import wraps, partial
import logging
# Utility decorator to attach a function as an attribute of obj
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    """
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    """
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)

        # Attach setter functions
        @attach_wrapper(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper

    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')

```

下面是交互环境下的使用例子:

```

>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> add(2, 3)
DEBUG:__main__:add
5
>>> # Change the Log message
>>> add.set_message('Add called')
>>> add(2, 3)
DEBUG:__main__:Add called
5
>>> # Change the Log Level
>>> add.set_level(logging.WARNING)
>>> add(2, 3)
WARNING:__main__:Add called
5
>>>

```

讨论

这一小节的关键点在于访问函数(如 `set_message()` 和 `set_level()`), 它们被作为属性赋给包装器。每个访问函数允许使用 `nonlocal` 来修改函数内部的变量。

还有一个令人吃惊的地方是访问函数会在多层装饰器间传播(如果你的装饰器都使用了 `@functools.wraps` 注解)。例如, 假设你引入另外一个装饰器, 比如9.2小节中的 `@timethis`, 像下面这样:

```

@timethis
@logged(logging.DEBUG)
def countdown(n):
    while n > 0:
        n -= 1

```

你会发现访问函数依旧有效:

```

>>> countdown(10000000)
DEBUG:__main__:countdown
countdown 0.8198461532592773
>>> countdown.set_level(logging.WARNING)
>>> countdown.set_message("Counting down to zero")
>>> countdown(10000000)
WARNING:__main__:Counting down to zero
countdown 0.8225970268249512
>>>

```

你还会发现即使装饰器像下面这样以相反的方向排放，效果也是一样的：

```
@logged(logging.DEBUG)
@timethis
def countdown(n):
    while n > 0:
        n -= 1
```

还能通过使用lambda表达式代码来让访问函数的返回不同的设定值：

```
@attach_wrapper(wrapper)
def get_level():
    return level

# Alternative
wrapper.get_level = lambda: level
```

一个比较难理解的地方就是对于访问函数的首次使用。例如，你可能会考虑另外一个方法直接访问函数的属性，如下：

```
@wraps(func)
def wrapper(*args, **kwargs):
    wrapper.log.log(wrapper.level, wrapper.logmsg)
    return func(*args, **kwargs)

# Attach adjustable attributes
wrapper.level = level
wrapper.logmsg = logmsg
wrapper.log = log
```

这个方法也可能正常工作，但前提是它必须是最外层的装饰器才行。如果它的上面还有另外的装饰器(比如上面提到的 `@timethis` 例子)，那么它会隐藏底层属性，使得修改它们没有任何作用。而通过使用访问函数就能避免这样的局限性。

最后提一点，这一小节的方案也可以作为9.9小节中装饰器类的另一种实现方法。

9.6 带可选参数的装饰器

问题

你想写一个装饰器，既可以传参数给它，比如 `@decorator`，也可以传递可选参数给它，比如 `@decorator(x,y,z)`。

解决方案

下面是9.5小节中日志装饰器的一个修改版本：

```
from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)

    logname = name if name else func.__module__
    log = logging.getLogger(logname)
    logmsg = message if message else func.__name__

    @wraps(func)
    def wrapper(*args, **kwargs):
        log.log(level, logmsg)
        return func(*args, **kwargs)

    return wrapper

# Example use
@logged
def add(x, y):
    return x + y

@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

可以看到，`@logged` 装饰器可以同时不带参数或带参数。

讨论

这里提到的这个问题就是通常所说的编程一致性问题。当我们使用装饰器的时候，大部分程序员习惯了要么不给它们传递任何参数，要么给它们传递确切参数。其实从技术上来讲，我们可以定义一个所有参数都是可选的装饰器，就像下面这样：

```
@logged()
def add(x, y):
    return x+y
```

但是，这种写法并不符合我们的习惯，有时候程序员忘记加上后面的括号会导致错误。这里我们向你展示了如何以一致的编程风格来同时满足没有括号和有括号两种情况。

为了理解代码是如何工作的，你需要非常熟悉装饰器是如何作用到函数上以及它们的调用规则。对于一个像下面这样的简单装饰器：

```
# Example use
@logged
def add(x, y):
    return x + y
```

这个调用序列跟下面等价：

```
def add(x, y):
    return x + y

add = logged(add)
```

这时候，被装饰函数会被当做第一个参数直接传递给 `logged` 装饰器。因此，`logged()` 中的第一个参数就是被包装函数本身。所有其他参数都必须有默认值。

而对于一个下面这样有参数的装饰器：

```
@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

调用序列跟下面等价：

```
def spam():
    print('Spam!')
spam = logged(level=logging.CRITICAL, name='example')(spam)
```

初始调用 `logged()` 函数时，被包装函数并没有传递进来。因此在装饰器内，它必须是可选的。这个反过来会迫使其他参数必须使用关键字来指定。并且，但这些参数被传递进来后，装饰器要返回一个接受一个函数参数并包装它的函数(参考9.5小节)。为了这样做，我们使用了一个技巧，就是利用 `functools.partial`。它会返回一个未完全初始化的自身，除了被包装函数外其他参数都已经确定下来了。可以参考7.8小节获取更多 `partial()` 方法的知识。

9.7 利用装饰器强制函数上的类型检查

问题

作为某种编程规约，你想在对函数参数进行强制类型检查。

解决方案

在演示实际代码前，先说明我们的目标：能对函数参数类型进行断言，类似下面这样：

```
>>> @typeassert(int, int)
... def add(x, y):
...     return x + y
...
>>>
>>> add(2, 3)
5
>>> add(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument y must be <class 'int'>
>>>
```

下面是使用装饰器技术来实现 `@typeassert`：

```

from inspect import signature
from functools import wraps

def typeassert(*ty_args, **ty_kwargs):
    def decorate(func):
        # If in optimized mode, disable type checking
        if not __debug__:
            return func

        # Map function argument names to supplied types
        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

        @wraps(func)
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            # Enforce type assertions across supplied arguments
            for name, value in bound_values.arguments.items():
                if name in bound_types:
                    if not isinstance(value, bound_types[name]):
                        raise TypeError(
                            'Argument {} must be {}'.format(name, bound_types[name])
                        )
            return func(*args, **kwargs)
        return wrapper
    return decorate

```

可以看出这个装饰器非常灵活，既可以指定所有参数类型，也可以只指定部分。并且可以通过位置或关键字来指定参数类型。下面是使用示例：

```

>>> @typeassert(int, z=int)
... def spam(x, y, z=42):
...     print(x, y, z)
...
>>> spam(1, 2, 3)
1 2 3
>>> spam(1, 'hello', 3)
1 hello 3
>>> spam(1, 'hello', 'world')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "contract.py", line 33, in wrapper
TypeError: Argument z must be <class 'int'>
>>>

```

讨论

这节是高级装饰器示例，引入了很多重要的概念。

首先，装饰器只在函数定义时被调用一次。有时候你去掉装饰器的功能，那么你只需要简单的返回被装饰函数即可。下面的代码中，如果全局变量 `__debug__` 被设置成了 `False`(当你使用 `-O` 或 `-OO` 参数的优化模式执行程序时)，那么就返回未修改过的函数本身：

```
def decorate(func):
    # If in optimized mode, disable type checking
    if not __debug__:
        return func
```

其次，这里还对被包装函数的参数签名进行了检查，我们使用了 `inspect.signature()` 函数。简单来讲，它运行你提取一个可调用对象的参数签名信息。例如：

```
>>> from inspect import signature
>>> def spam(x, y, z=42):
...     pass
...
>>> sig = signature(spam)
>>> print(sig)
(x, y, z=42)
>>> sig.parameters
mappingproxy(OrderedDict([('x', <Parameter at 0x10077a050 'x'>),
('y', <Parameter at 0x10077a158 'y'>), ('z', <Parameter at 0x10077a1b0 'z'>)]))
>>> sig.parameters['z'].name
'z'
>>> sig.parameters['z'].default
42
>>> sig.parameters['z'].kind
<ParameterKind: 'POSITIONAL_OR_KEYWORD'>
>>>
```

装饰器的开始部分，我们使用了 `bind_partial()` 方法来执行从指定类型到名称的部分绑定。下面是例子演示：

```
>>> bound_types = sig.bind_partial(int, z=int)
>>> bound_types
<inspect.BoundArguments object at 0x10069bb50>
>>> bound_types.arguments
OrderedDict([('x', <class 'int'>), ('z', <class 'int'>)])
>>>
```

在这个部分绑定中，你可以注意到缺失的参数被忽略了(比如并没有对y进行绑定)。不过最重要的是创建了一个有序字典 `bound_types.arguments`。这个字典会将参数名以函数签名中相同顺序映射到指定的类型值上面去。在我们的装饰器例子中，这个映射包含了我们要强制指定的类型断言。

在装饰器创建的实际包装函数中使用到了 `sig.bind()` 方法。`bind()` 跟 `bind_partial()` 类似，但是它不允许忽略任何参数。因此有了下面的结果：

```
>>> bound_values = sig.bind(1, 2, 3)
>>> bound_values.arguments
OrderedDict([('x', 1), ('y', 2), ('z', 3)])
>>>
```

使用这个映射我们可以很轻松的实现我们的强制类型检查：

```
>>> for name, value in bound_values.arguments.items():
...     if name in bound_types.arguments:
...         if not isinstance(value, bound_types.arguments[name]):
...             raise TypeError()
...
>>>
```

不过这个方案还有点小瑕疵，它对于有默认值的参数并不适用。比如下面的代码可以正常工作，尽管items的类型是错误的：

```
>>> @typeassert(int, list)
... def bar(x, items=None):
...     if items is None:
...         items = []
...         items.append(x)
...     return items
>>> bar(2)
[2]
>>> bar(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument items must be <class 'list'>
>>> bar(4, [1, 2, 3])
[1, 2, 3, 4]
>>>
```

最后一点是关于适用装饰器参数和函数注解之间的争论。例如，为什么不像下面这样写一个装饰器来查找函数中的注解呢？

```
@typeassert
def spam(x:int, y, z:int = 42):
    print(x,y,z)
```

一个可能的原因是如果使用了函数参数注解，那么就被限制了。如果注解被用来做类型检查就不能做其他事情了。而且 `@typeassert` 不能再用于使用注解做其他事情的函数了。而使用上面的装饰器参数灵活性大多了，也更加通用。

可以在PEP 362以及 `inspect` 模块中找到更多关于函数参数对象的信息。在9.16小节还有另外一个例子。

9.8 将装饰器定义为类的一部分

问题

你想在类中定义装饰器，并将其作用在其他函数或方法上。

解决方案

在类里面定义装饰器很简单，但是你首先要确认它的使用方式。比如到底是作为一个实例方法还是类方法。下面我们用例子来阐述它们的不同：

```
from functools import wraps

class A:
    # Decorator as an instance method
    def decorator1(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 1')
            return func(*args, **kwargs)
        return wrapper

    # Decorator as a class method
    @classmethod
    def decorator2(cls, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 2')
            return func(*args, **kwargs)
        return wrapper
```

下面是一使用例子：

```
# As an instance method
a = A()
@a.decorator1
def spam():
    pass

# As a class method
@A.decorator2
def grok():
    pass
```

仔细观察可以发现一个是实例调用，一个是类调用。

讨论

在类中定义装饰器初看上去好像很奇怪，但是在标准库中有很多这样的例子。特别的，`@property` 装饰器实际上是一个类，它里面定义了三个方法

`getter()`，`setter()`，`deleter()`，每一个方法都是一个装饰器。例如：

```
class Person:
    # Create a property instance
    first_name = property()

    # Apply decorator methods
    @first_name.getter
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
```

它为什么要这么定义的主要原因是各种不同的装饰器方法会在关联的 `property` 实例上操作它的状态。因此，任何时候只要你碰到需要在装饰器中记录或绑定信息，那么这不失为一种可行方法。

在类中定义装饰器有个难理解的地方就是对于额外参数 `self` 或 `cls` 的正确使用。尽管最外层的装饰器函数比如 `decorator1()` 或 `decorator2()` 需要提供一个 `self` 或 `cls` 参数，但是在两个装饰器内部被创建的 `wrapper()` 函数并不需要包含这个 `self` 参数。你唯一需要这个参数是在你确实要访问包装器中这个实例的某些部分的时候。其他情况下都不用去管它。

对于类里面定义的包装器还有一点比较难理解，就是在涉及到继承的时候。例如，假设你想让在A中定义的装饰器作用在子类B中。你需要像下面这样写：

```
class B(A):
    @A.decorator2
    def bar(self):
        pass
```

也就是说，装饰器要被定义成类方法并且你必须显式的使用父类名去调用它。你不能使用 `@B.decorator2`，因为在方法定义时，这个类B还没有被创建。

9.9 将装饰器定义为类

问题

你想使用一个装饰器去包装函数，但是希望返回一个可调用的实例。你需要让你的装饰器可以同时工作在类定义的内部和外部。

解决方案

为了将装饰器定义成一个实例，你需要确保它实现了 `__call__()` 和 `__get__()` 方法。例如，下面的代码定义了一个类，它在其他函数上放置一个简单的记录层：

```
import types
from functools import wraps

class Profiled:
    def __init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def __call__(self, *args, **kwargs):
        self.ncalls += 1
        return self.__wrapped__(*args, **kwargs)

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return types.MethodType(self, instance)
```

你可以将它当做一个普通的装饰器来使用，在类里面或外面都可以：

```
@Profiled
def add(x, y):
    return x + y

class Spam:
    @Profiled
    def bar(self, x):
        print(self, x)
```

在交互环境中的使用示例：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls
2
>>> s = Spam()
>>> s.bar(1)
<__main__.Spam object at 0x10069e9d0> 1
>>> s.bar(2)
<__main__.Spam object at 0x10069e9d0> 2
>>> s.bar(3)
<__main__.Spam object at 0x10069e9d0> 3
>>> Spam.bar.ncalls
3
```

讨论

将装饰器定义成类通常是很简单的。但是这里还是有一些细节需要解释下，特别是当你将它作用在实例方法上的时候。

首先，使用 `functools.wraps()` 函数的作用跟之前还是一样，将被包装函数的元信息复制到可调用实例中去。

其次，通常很容易会忽视上面的 `__get__()` 方法。如果你忽略它，保持其他代码不变再次运行，你会发现当你去调用被装饰实例方法时出现很奇怪的问题。例如：

```
>>> s = Spam()
>>> s.bar(3)
Traceback (most recent call last):
...
TypeError: bar() missing 1 required positional argument: 'x'
```

出错原因是当方法函数在一个类中被查找时，它们的 `__get__()` 方法依据描述器协议被调用，在8.9小节已经讲述过描述器协议了。在这里，`__get__()` 的目的是创建一个绑定方法对象(最终会给这个方法传递`self`参数)。下面是一个例子来演示底层原理：

```
>>> s = Spam()
>>> def grok(self, x):
...     pass
...
>>> grok.__get__(s, Spam)
<bound method Spam.grok of <__main__.Spam object at 0x100671e90>>
>>>
```

`__get__()` 方法是为了确保绑定方法对象能被正确的创建。`type.MethodType()` 手动创建一个绑定方法来使用。只有当实例被使用的时候绑定方法才会被创建。如果这个方法是在类上面来访问，那么 `__get__()` 中的`instance`参数会被设置成`None`并直接返回 `Profiled` 实例本身。这样的话我们就可以提取它的 `ncalls` 属性了。

如果你想避免一些混乱，也可以考虑另外一个使用闭包和 `nonlocal` 变量实现的装饰器，这个在9.5小节有讲到。例如：

```
import types
from functools import wraps

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper

# Example
@profiled
def add(x, y):
    return x + y
```

这个方式跟之前的效果几乎一样，除了对于 `ncalls` 的访问现在是通过一个被绑定为属性的函数来实现，例如：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls()
2
>>>
```

9.10 为类和静态方法提供装饰器

问题

你想给类或静态方法提供装饰器。

解决方案

给类或静态方法提供装饰器是很简单的，不过要确保装饰器在 `@classmethod` 或 `@staticmethod` 之前。例如：

```

import time
from functools import wraps

# A simple decorator
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(end-start)
        return r
    return wrapper

# Class illustrating application of the decorator to different kinds of methods
class Spam:
    @timethis
    def instance_method(self, n):
        print(self, n)
        while n > 0:
            n -= 1

    @classmethod
    @timethis
    def class_method(cls, n):
        print(cls, n)
        while n > 0:
            n -= 1

    @staticmethod
    @timethis
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1

```

装饰后的类和静态方法可正常工作，只不过增加了额外的计时功能：

```

>>> s = Spam()
>>> s.instance_method(1000000)
<__main__.Spam object at 0x1006a6050> 1000000
0.11817407608032227
>>> Spam.class_method(1000000)
<class '__main__.Spam'> 1000000
0.11334395408630371
>>> Spam.static_method(1000000)
1000000
0.11740279197692871
>>>

```

讨论

如果你把装饰器的顺序写错了就会出错。例如，假设你像下面这样写：

```
class Spam:
    @timethis
    @staticmethod
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

那么你调用这个镜头方法时就会报错：

```
>>> Spam.static_method(1000000)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "timethis.py", line 6, in wrapper
start = time.time()
TypeError: 'staticmethod' object is not callable
>>>
```

问题在于 `@classmethod` 和 `@staticmethod` 实际上并不会创建可直接调用的对象，而是创建特殊的描述器对象(参考8.9小节)。因此当你试着在其他装饰器中将它们当做函数来使用时就会出错。确保这种装饰器出现在装饰器链中的第一个位置可以修复这个问题。

当我们在抽象基类中定义类方法和静态方法(参考8.12小节)时，这里讲到的知识就很有用了。例如，如果你想定义一个抽象类方法，可以使用类似下面的代码：

```
from abc import ABCMeta, abstractmethod
class A(metaclass=ABCMeta):
    @classmethod
    @abstractmethod
    def method(cls):
        pass
```

在这段代码中，`@classmethod` 跟 `@abstractmethod` 两者的顺序是有讲究的，如果你调换它们的顺序就会出错。

9.11 装饰器为被包装函数增加参数

问题

你想在装饰器中给被包装函数增加额外的参数，但是不能影响这个函数现有的调用规则。

解决方案

可以使用关键字参数来给被包装函数增加额外参数。考虑下面的装饰器：

```
from functools import wraps

def optional_debug(func):
    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    return wrapper
```

```
>>> @optional_debug
... def spam(a,b,c):
...     print(a,b,c)
...
>>> spam(1,2,3)
1 2 3
>>> spam(1,2,3, debug=True)
Calling spam
1 2 3
>>>
```

讨论

通过装饰器来给被包装函数增加参数的做法并不常见。尽管如此，有时候它可以避免一些重复代码。例如，如果你有下面这样的代码：

```

def a(x, debug=False):
    if debug:
        print('Calling a')

def b(x, y, z, debug=False):
    if debug:
        print('Calling b')

def c(x, y, debug=False):
    if debug:
        print('Calling c')

```

那么你可以将其重构成这样：

```

from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper

@optional_debug
def a(x):
    pass

@optional_debug
def b(x, y, z):
    pass

@optional_debug
def c(x, y):
    pass

```

这种实现方案之所以行得通，在于强制关键字参数很容易被添加到接受 `*args` 和 `**kwargs` 参数的函数中。通过使用强制关键字参数，它被作为一个特殊情况被挑选出来，并且接下来仅仅使用剩余的位置和关键字参数去调用这个函数时，这个特殊参数会被排除在外。也就是说，它并不会被纳入到 `**kwargs` 中去。

还有一个难点就是如何去处理被添加的参数与被包装函数参数直接的名字冲突。例如，如果装饰器 `@optional_debug` 作用在一个已经拥有一个 `debug` 参数的函数上时会有问题。这里我们增加了一步名字检查。

上面的方案还可以更完美一点，因为精明的程序员应该发现了被包装函数的函数签名其实是错误的。例如：

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> import inspect
>>> print(inspect.signature(add))
(x, y)
>>>
```

通过如下的修改，可以解决这个问题：

```
from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    sig = inspect.signature(func)
    parms = list(sig.parameters.values())
    parms.append(inspect.Parameter('debug',
                                   inspect.Parameter.KEYWORD_ONLY,
                                   default=False))
    wrapper.__signature__ = sig.replace(parameters=parms)
    return wrapper
```

通过这样的修改，包装后的函数签名就能正确的显示 `debug` 参数的存在了。例如：

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> print(inspect.signature(add))
(x, y, *, debug=False)
>>> add(2,3)
5
>>>
```

参考9.16小节获取更多关于函数签名的信息。

9.12 使用装饰器扩充类的功能

问题

你想通过反省或者重写类定义的某部分来修改它的行为，但是你不希望使用继承或元类的方式。

解决方案

这种情况可能是类装饰器最好的使用场景了。例如，下面是一个重写了特殊方法

`__getattr__` 的类装饰器，可以打印日志：

```
def log_getattribute(cls):
    # Get the original implementation
    orig_getattribute = cls.__getattr__

    # Make a new definition
    def new_getattribute(self, name):
        print('getting:', name)
        return orig_getattribute(self, name)

    # Attach to the class and return
    cls.__getattr__ = new_getattribute
    return cls

# Example use
@log_getattribute
class A:
    def __init__(self, x):
        self.x = x
    def spam(self):
        pass
```

下面是使用效果：

```
>>> a = A(42)
>>> a.x
getting: x
42
>>> a.spam()
getting: spam
>>>
```

讨论

类装饰器通常可以作为其他高级技术比如混入或元类的一种非常简洁的替代方案。比如，上面示例中的另外一种实现使用到继承：

```
class LoggedGetattribute:
    def __getattribute__(self, name):
        print('getting:', name)
        return super().__getattribute__(name)

# Example:
class A(LoggedGetattribute):
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

这种方案也行得通，但是为了去理解它，你就必须知道方法调用顺序、`super()` 以及其它8.7小节介绍的继承知识。某种程度上来讲，类装饰器方案就显得更加直观，并且它不会引入新的继承体系。它的运行速度也更快一些，因为他并不依赖 `super()` 函数。

如果你系想在一个类上面使用多个类装饰器，那么就需要注意下顺序问题。例如，一个装饰器A会将其装饰的方法完整替换成另一种实现，而另一个装饰器B只是简单的在其装饰的方法中添加点额外逻辑。那么这时候装饰器A就需要放在装饰器B的前面。

你还可以回顾一下8.13小节另外一个关于类装饰器的有用的例子。

9.13 使用元类控制实例的创建

问题

你想通过改变实例创建方式来实现单例、缓存或其他类似的特性。

解决方案

Python程序员都知道，如果你定义了一个类，就能像函数一样的调用它来创建实例，例如：

```
class Spam:
    def __init__(self, name):
        self.name = name

a = Spam('Guido')
b = Spam('Diana')
```

如果你想自定义这个步骤，你可以定义一个元类并自己实现 `__call__()` 方法。

为了演示，假设你不想任何人创建这个类的实例：

```
class NoInstances(type):
    def __call__(self, *args, **kwargs):
        raise TypeError("Can't instantiate directly")

# Example
class Spam(metaclass=NoInstances):
    @staticmethod
    def grok(x):
        print('Spam.grok')
```

这样的话，用户只能调用这个类的静态方法，而不能使用通常的方法来创建它的实例。例如：

```
>>> Spam.grok(42)
Spam.grok
>>> s = Spam()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example1.py", line 7, in __call__
    raise TypeError("Can't instantiate directly")
TypeError: Can't instantiate directly
>>>
```

现在，假如你想实现单例模式（只能创建唯一实例的类），实现起来也很简单：

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            return self.__instance

# Example
class Spam(metaclass=Singleton):
    def __init__(self):
        print('Creating Spam')
```

那么Spam类就只能创建唯一的实例了，演示如下：

```
>>> a = Spam()
Creating Spam
>>> b = Spam()
>>> a is b
True
>>> c = Spam()
>>> a is c
True
>>>
```

最后，假设你想创建8.25小节中那样的缓存实例。下面我们可以通过元类来实现：

```

import weakref

class Cached(type):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__cache = weakref.WeakValueDictionary()

    def __call__(self, *args):
        if args in self.__cache:
            return self.__cache[args]
        else:
            obj = super().__call__(*args)
            self.__cache[args] = obj
            return obj

# Example
class Spam(metaclass=Cached):
    def __init__(self, name):
        print('Creating Spam({!r})'.format(name))
        self.name = name

```

然后我也来测试一下：

```

>>> a = Spam('Guido')
Creating Spam('Guido')
>>> b = Spam('Diana')
Creating Spam('Diana')
>>> c = Spam('Guido') # Cached
>>> a is b
False
>>> a is c # Cached value returned
True
>>>

```

讨论

利用元类实现多种实例创建模式通常要比不使用元类的方式优雅得多。

假设你不使用元类，你可能需要将类隐藏在某些工厂函数后面。比如为了实现一个单例，你你可能会像下面这样写：

```
class _Spam:
    def __init__(self):
        print('Creating Spam')

    _spam_instance = None

def Spam():
    global _spam_instance

    if _spam_instance is not None:
        return _spam_instance
    else:
        _spam_instance = _Spam()
        return _spam_instance
```

尽管使用元类可能会涉及到比较高级点的技术，但是它的代码看起来会更加简洁舒服，而且也更加直观。

更多关于创建缓存实例、弱引用等内容，请参考8.25小节。

9.14 捕获类的属性定义顺序

问题

你想自动记录一个类中属性和方法定义的顺序，然后可以利用它来做很多操作（比如序列化、映射到数据库等等）。

解决方案

利用元类可以很容易的捕获类的定义信息。下面是一个例子，使用了一个OrderedDict来记录描述器的定义顺序：

```

from collections import OrderedDict

# A set of descriptors for various types
class Typed:
    _expected_type = type(None)
    def __init__(self, name=None):
        self._name = name

    def __set__(self, instance, value):
        if not isinstance(value, self._expected_type):
            raise TypeError('Expected ' + str(self._expected_type))
        instance.__dict__[self._name] = value

class Integer(Typed):
    _expected_type = int

class Float(Typed):
    _expected_type = float

class String(Typed):
    _expected_type = str

# Metaclass that uses an OrderedDict for class body
class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        order = []
        for name, value in clsdict.items():
            if isinstance(value, Typed):
                value._name = name
                order.append(name)
        d['_order'] = order
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return OrderedDict()

```

在这个元类中，执行类主体时描述器的定义顺序会被一个

`OrderedDict` 捕获到，生成的有序名称从字典中提取出来并放入类属性 `__order` 中。这样的话类中的方法可以通过多种方式来使用它。例如，下面是一个简单的类，使用这个排序字典来实现将一个类实例的数据序列化为一行CSV数据：


```
class Structure(metaclass=OrderedMeta):
    def as_csv(self):
        return ','.join(str(getattr(self,name)) for name in self._order)

# Example use
class Stock(Structure):
    name = String()
    shares = Integer()
    price = Float()

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

我们在交互式环境中测试一下这个Stock类:

```
>>> s = Stock('GOOG',100,490.1)
>>> s.name
'GOOG'
>>> s.as_csv()
'GOOG,100,490.1'
>>> t = Stock('AAPL','a lot', 610.23)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dupmethod.py", line 34, in __init__
TypeError: shares expects <class 'int'>
>>>
```

讨论

本节一个关键点就是OrderedMeta元类中定义的`__prepare__()`方法。这个方法会在开始定义类和它的父类的时候被执行。它必须返回一个映射对象以便在类定义体中被使用到。我们这里通过返回了一个OrderedDict而不是一个普通的字典，可以很容易的捕获定义的顺序。

如果你想构造自己的类字典对象，可以很容易的扩展这个功能。比如，下面的这个修改方案可以防止重复的定义：

```

from collections import OrderedDict

class NoDupOrderedDict(OrderedDict):
    def __init__(self, clsname):
        self.clsname = clsname
        super().__init__()
    def __setitem__(self, name, value):
        if name in self:
            raise TypeError('{} already defined in {}'.format(name, self.clsname))
        super().__setitem__(name, value)

class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        d['_order'] = [name for name in clsdict if name[0] != '_']
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return NoDupOrderedDict(clsname)

```

下面我们测试重复的定义会出现什么情况：

```

>>> class A(metaclass=OrderedMeta):
...     def spam(self):
...         pass
...     def spam(self):
...         pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in A
  File "dupmethod2.py", line 25, in __setitem__
    (name, self.clsname))
TypeError: spam already defined in A
>>>

```

最后还有一点很重要，就是在 `__new__()` 方法中对于元类中被修改字典的处理。尽管类使用了另外一个字典来定义，在构造最终的 `class` 对象的时候，我们仍然需要将这个字典转换为一个正确的 `dict` 实例。通过语句 `d = dict(clsdict)` 来完成这个效果。

对于很多应用程序而已，能够捕获类定义的顺序是一个看似不起眼却又非常重要的特性。例如，在对象关系映射中，我们通常会看到下面这种方式定义的类：

```
class Stock(Model):
    name = String()
    shares = Integer()
    price = Float()
```

在框架底层，我们必须捕获定义的顺序来将对象映射到元组或数据库表中的行（就类似于上面例子中的 `as_csv()` 的功能）。这节演示的技术非常简单，并且通常会比其他类似方法（通常都要在描述器类中维护一个隐藏的计数器）要简单的多。

9.15 定义有可选参数的元类

问题

你想定义一个元类，允许类定义时提供可选参数，这样可以控制或配置类型的创建过程。

解决方案

在定义类的时候，Python允许我们使用 `__metaclass__` 关键字参数来指定特定的元类。例如使用抽象基类：

```
from abc import ABCMeta, abstractmethod
class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxsize=None):
        pass

    @abstractmethod
    def write(self, data):
        pass
```

然而，在自定义元类中我们还可以提供其他的关键字参数，如下所示：

```
class Spam(metaclass=MyMeta, debug=True, synchronize=True):
    pass
```

为了使元类支持这些关键字参数，你必须确保在 `__prepare__()`，`__new__()` 和 `__init__()` 方法中都使用强制关键字参数。就像下面这样：

```

class MyMeta(type):
    # Optional
    @classmethod
    def __prepare__(cls, name, bases, *, debug=False, synchronize=False):
        # Custom processing
        pass
        return super().__prepare__(name, bases)

    # Required
    def __new__(cls, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        pass
        return super().__new__(cls, name, bases, ns)

    # Required
    def __init__(self, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        pass
        super().__init__(name, bases, ns)

```

讨论

给一个元类添加可选关键字参数需要你完全弄懂类创建的所有步骤，因为这些参数会被传递给每一个相关的方法。`__prepare__()` 方法在所有类定义开始执行前首先被调用，用来创建类命名空间。通常来讲，这个方法只是简单的返回一个字典或其他映射对象。

`__new__()` 方法被用来实例化最终的类对象。它在类的主体被执行完后开始执行。

`__init__()` 方法最后被调用，用来执行其他的一些初始化工作。

当我们构造元类的时候，通常只需要定义一个 `__new__()` 或 `__init__()` 方法，但不是两个都定义。但是，如果需要接受其他的关键字参数的话，这两个方法就要同时提供，并且都要提供对应的参数签名。默认的 `__prepare__()` 方法接受任意的关键字参数，但是会忽略它们，所以只有当这些额外的参数可能会影响到类命名空间的创建时你才需要去定义 `__prepare__()` 方法。

通过使用强制关键字参数，在类的创建过程中我们必须通过关键字来指定这些参数。

使用关键字参数配置一个元类还可以视作对类变量的一种替代方式。例如：

```

class Spam(metaclass=MyMeta):
    debug = True
    synchronize = True
    pass

```

将这些属性定义为参数的好处在于它们不会污染类的名称空间，这些属性仅仅只从属于类的创建阶段，而不是类中的语句执行阶段。另外，它们在 `__prepare__()` 方法中是可以被访问的，因为这个方法会在所有类主体执行前被执行。但是类变量只能在元类的 `__new__()` 和 `__init__()` 方法中可见。

9.16 *args和**kwargs的强制参数签名

问题

你有一个函数或方法，它使用*args和**kwargs作为参数，这样使得它比较通用，但有时候你想检查传递进来的参数是不是某个你想要的类型。

解决方案

对任何涉及到操作函数调用签名的问题，你都应该使用 `inspect` 模块中的签名特性。我们最主要关注两个类：`Signature` 和 `Parameter`。下面是一个创建函数前面的交互例子：

```
>>> from inspect import Signature, Parameter
>>> # Make a signature for a func(x, y=42, *, z=None)
>>> parms = [ Parameter('x', Parameter.POSITIONAL_OR_KEYWORD),
...           Parameter('y', Parameter.POSITIONAL_OR_KEYWORD, default=42),
...           Parameter('z', Parameter.KEYWORD_ONLY, default=None) ]
>>> sig = Signature(parms)
>>> print(sig)
(x, y=42, *, z=None)
>>>
```

一旦你有了一个签名对象，你就可以使用它的 `bind()` 方法很容易的将它绑定到 `*args` 和 `**kwargs` 上去。下面是一个简单的演示：

```

>>> def func(*args, **kwargs):
...     bound_values = sig.bind(*args, **kwargs)
...     for name, value in bound_values.arguments.items():
...         print(name,value)
...
>>> # Try various examples
>>> func(1, 2, z=3)
x 1
y 2
z 3
>>> func(1)
x 1
>>> func(1, z=3)
x 1
z 3
>>> func(y=2, x=1)
x 1
y 2
>>> func(1, 2, 3, 4)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1972, in _bind
    raise TypeError('too many positional arguments')
TypeError: too many positional arguments
>>> func(y=2)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1961, in _bind
    raise TypeError(msg) from None
TypeError: 'x' parameter lacking default value
>>> func(1, y=2, x=3)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1985, in _bind
    '{arg!r}'.format(arg=param.name))
TypeError: multiple values for argument 'x'
>>>

```

可以看出来，通过将签名和传递的参数绑定起来，可以强制函数调用遵循特定的规则，比如必填、默认、重复等等。

下面是一个强制函数签名更具体的例子。在代码中，我们在基类中先定义了一个非常通用的 `__init__()` 方法，然后我们强制所有的子类必须提供一个特定的参数签名。

```

from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
              for name in names]
    return Signature(parms)

class Structure:
    __signature__ = make_sig()
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Example use
class Stock(Structure):
    __signature__ = make_sig('name', 'shares', 'price')

class Point(Structure):
    __signature__ = make_sig('x', 'y')

```

下面是使用这个 `Stock` 类的示例：

```

>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> s1 = Stock('ACME', 100, 490.1)
>>> s2 = Stock('ACME', 100)
Traceback (most recent call last):
...
TypeError: 'price' parameter lacking default value
>>> s3 = Stock('ACME', 100, 490.1, shares=50)
Traceback (most recent call last):
...
TypeError: multiple values for argument 'shares'
>>>

```

讨论

在我们需要构建通用函数库、编写装饰器或实现代理的时候，对于 `*args` 和 `**kwargs` 的使用是很普遍的。但是，这样的函数有一个缺点就是当你想要实现自己的参数检验时，代码就会笨拙混乱。在8.11小节里面有这样一个例子。这时候我们可以通过一个签名对象来简化它。

在最后的方案实例中，我们还可以通过使用自定义元类来创建签名对象。下面演示怎样来实现：

```

from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
              for name in names]
    return Signature(parms)

class StructureMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields', []))
        return super().__new__(cls, clsname, bases, clsdict)

class Structure(metaclass=StructureMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Example
class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']

```

当我们自定义签名的时候，将签名存储在特定的属性 `__signature__` 中通常是很有用的。这样的话，在使用 `inspect` 模块执行内省的代码就能发现签名并将它作为调用约定。

```

>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> print(inspect.signature(Point))
(x, y)
>>>

```

9.17 在类上强制执行编程规约

问题

你的程序包含一个很大的类继承体系，你希望强制执行某些编程规约（或者代码诊断）来帮助程序员保持清醒。

解决方案

如果你想监控类的定义，通常可以通过定义一个元类。一个基本元类通常是继承自 `type` 并重定义它的 `__new__()` 方法 或者是 `__init__()` 方法。比如：

```
class MyMeta(type):
    def __new__(self, clsname, bases, clsdict):
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
        return super().__new__(cls, clsname, bases, clsdict)
```

另一种是，定义 `__init__()` 方法：

```
class MyMeta(type):
    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
```

为了使用这个元类，你通常要将它放到到一个顶级父类定义中，然后其他的类继承这个顶级父类。例如：

```
class Root(metaclass=MyMeta):
    pass

class A(Root):
    pass

class B(Root):
    pass
```

元类的一个关键特点是它允许你在定义的时候检查类的内容。在重新定义 `__init__()` 方法中，你可以很轻松的检查类字典、父类等等。并且，一旦某个元类被指定给了某个类，那么就会被继承到所有子类中去。因此，一个框架的构建者就能在大型的继承体系中通过给一个顶级父类指定一个元类去捕获所有下面子类的定义。

作为一个具体的应用例子，下面定义了一个元类，它会拒绝任何有混合大小写名字作为方法的类定义（可能是想气死Java程序员^^）：

```
class NoMixedCaseMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        for name in clsdict:
            if name.lower() != name:
                raise TypeError('Bad attribute name: ' + name)
        return super().__new__(cls, clsname, bases, clsdict)

class Root(metaclass=NoMixedCaseMeta):
    pass

class A(Root):
    def foo_bar(self): # Ok
        pass

class B(Root):
    def fooBar(self): # TypeError
        pass
```

作为更高级和实用的例子，下面有一个元类，它用来检测重载方法，确保它的调用参数跟父类中原始方法有着相同的参数签名。

```

from inspect import signature
import logging

class MatchSignaturesMeta(type):

    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        sup = super(self, self)
        for name, value in clsdict.items():
            if name.startswith('_') or not callable(value):
                continue
            # Get the previous definition (if any) and compare the signatures
            prev_dfn = getattr(sup, name, None)
            if prev_dfn:
                prev_sig = signature(prev_dfn)
                val_sig = signature(value)
                if prev_sig != val_sig:
                    logging.warning('Signature mismatch in %s. %s != %s',
                                    value.__qualname__, prev_sig, val_sig)

# Example
class Root(metaclass=MatchSignaturesMeta):
    pass

class A(Root):
    def foo(self, x, y):
        pass

    def spam(self, x, *, z):
        pass

# Class with redefined methods, but slightly different signatures
class B(A):
    def foo(self, a, b):
        pass

    def spam(self, x, z):
        pass

```

如果你运行这段代码，就会得到下面这样的输出结果：

```

WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (self, x, z)
WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self, a, b)

```

这种警告信息对于捕获一些微妙的程序bug是很有用的。例如，如果某个代码依赖于传递给方法的关键字参数，那么当子类改变参数名字的时候就会调用出错。

讨论

在大型面向对象的程序中，通常将类的定义放在元类中控制是很有用的。元类可以监控类的定义，警告编程人员某些没有注意到的可能出现的问题。

有人可能会说，像这样的错误可以通过程序分析工具或IDE去做会更好些。诚然，这些工具是很有用。但是，如果你在构建一个框架或函数库供其他人使用，那么你没办法去控制使用者要使用什么工具。因此，对于这种类型的程序，如果可以在元类中做检测或许可以带来更好的用户体验。

在元类中选择重新定义 `__new__()` 方法还是 `__init__()` 方法取决于你想怎样使用结果类。`__new__()` 方法在类创建之前被调用，通常用于通过某种方式（比如通过改变类字典的内容）修改类的定义。而 `__init__()` 方法是在类被创建之后被调用，当你需要完整构建类对象的时候会很有用。在最后一个例子中，这是必要的，因为它使用了 `super()` 函数来搜索之前的定义。它只能在类的实例被创建之后，并且相应的方法解析顺序也已经被设置好了。

最后一个例子还演示了Python的函数签名对象的使用。实际上，元类会管理中每个一个调用定义，搜索前一个定义（如果有的话），然后通过使用 `inspect.signature()` 来简单的比较它们的调用签名。

最后一点，代码中有一行使用了 `super(self, self)` 并不是排版错误。当使用元类的时候，我们要时刻记住一点就是 `self` 实际上是一个类对象。因此，这条语句其实就是用来寻找位于继承体系中构建 `self` 父类的定义。

9.18 以编程方式定义类

问题

你在写一段代码，最终需要创建一个新的类对象。你考虑将类的定义源代码以字符串的形式发布出去。并且使用函数比如 `exec()` 来执行它，但是你想寻找一个更加优雅的解决方案。

解决方案

你可以使用函数 `types.new_class()` 来初始化新的类对象。你需要做的只是提供类的名字、父类元组、关键字参数，以及一个用成员变量填充类字典的回调函数。例如：

```

# stock.py
# Example of making a class manually from parts

# Methods
def __init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price
def cost(self):
    return self.shares * self.price

cls_dict = {
    '__init__': __init__,
    'cost': cost,
}

# Make a class
import types

Stock = types.new_class('Stock', (), {}, lambda ns: ns.update(cls_dict))
Stock.__module__ = __name__

```

这种方式会构建一个普通的类对象，并且按照你的期望工作：

```

>>> s = Stock('ACME', 50, 91.1)
>>> s
<stock.Stock object at 0x1006a9b10>
>>> s.cost()
4555.0
>>>

```

这种方法中，一个比较难理解的地方是在调用完 `types.new_class()` 对 `Stock.__module__` 的赋值。每次当一个类被定义后，它的 `__module__` 属性包含定义它的模块名。这个名字用于生成 `__repr__()` 方法的输出。它同样也被用于很多库，比如 `pickle`。因此，为了让你创建的类是“正确”的，你需要确保这个属性也设置正确了。

如果你想创建的类需要一个不同的元类，可以通过 `types.new_class()` 第三个参数传递给它。例如：

```

>>> import abc
>>> Stock = types.new_class('Stock', (), {'metaclass': abc.ABCMeta},
...                               lambda ns: ns.update(cls_dict))
...
>>> Stock.__module__ = __name__
>>> Stock
<class '__main__.Stock'>
>>> type(Stock)
<class 'abc.ABCMeta'>
>>>

```

第三个参数还可以包含其他的关键字参数。比如，一个类的定义如下：

```

class Spam(Base, debug=True, typecheck=False):
    pass

```

那么可以将其翻译成如下的 `new_class()` 调用形式：

```

Spam = types.new_class('Spam', (Base,),
                      {'debug': True, 'typecheck': False},
                      lambda ns: ns.update(cls_dict))

```

`new_class()` 第四个参数最神秘，它是一个用来接受类命名空间的映射对象的函数。通常这是一个普通的字典，但是它实际上是 `__prepare__()` 方法返回的任意对象，这个在9.14小节已经介绍过了。这个函数需要使用上面演示的 `update()` 方法给命名空间增加内容。

讨论

很多时候如果能构造新的类对象是很有用的。有个很熟悉的例子是调用

`collections.namedtuple()` 函数，例如：

```

>>> Stock = collections.namedtuple('Stock', ['name', 'shares', 'price'])
>>> Stock
<class '__main__.Stock'>
>>>

```

`namedtuple()` 使用 `exec()` 而不是上面介绍的技术。但是，下面通过一个简单的变化，我们直接创建一个类：

```

import operator
import types
import sys

def named_tuple(classname, fieldnames):
    # Populate a dictionary of field property accessors
    cls_dict = { name: property(operator.itemgetter(n))
                 for n, name in enumerate(fieldnames) }

    # Make a __new__ function and add to the class dict
    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        return tuple.__new__(cls, args)

    cls_dict['__new__'] = __new__

    # Make the class
    cls = types.new_class(classname, (tuple,), {},
                          lambda ns: ns.update(cls_dict))

    # Set the module to that of the caller
    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls

```

这段代码的最后部分使用了一个所谓的“框架魔法”，通过调用 `sys._getframe()` 来获取调用者的模块名。另外一个框架魔法例子在2.15小节中有介绍过。

下面的例子演示了前面的代码是如何工作的：

```

>>> Point = named_tuple('Point', ['x', 'y'])
>>> Point
<class '__main__.Point'>
>>> p = Point(4, 5)
>>> len(p)
2
>>> p.x
4
>>> p.y
5
>>> p.x = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> print('%s %s' % p)
4 5
>>>

```

这项技术一个很重要的方面是它对于元类的正确使用。你可能像通过直接实例化一个元类来直接创建一个类：

```
Stock = type('Stock', (), cls_dict)
```

这种方法的问题在于它忽略了一些关键步骤，比如对于元类中 `__prepare__()` 方法的调用。通过使用 `types.new_class()`，你可以保证所有的必要初始化步骤都能得到执行。比如，`types.new_class()` 第四个参数的回调函数接受 `__prepare__()` 方法返回的映射对象。

如果你仅仅只是想执行准备步骤，可以使用 `types.prepare_class()`。例如：

```
import types
metaclass, kwargs, ns = types.prepare_class('Stock', (), {'metaclass': type})
```

它会查找合适的元类并调用它的 `__prepare__()` 方法。然后这个元类保存它的关键字参数，准备命名空间后被返回。

更多信息, 请参考 [PEP 3115](#), 以及 [Python documentation](#) .

9.19 在定义的时候初始化类的成员

问题

你想在类被定义的时候就初始化一部分类的成员，而不是要等到实例被创建后。

解决方案

在类定义时就执行初始化或设置操作是元类的一个典型应用场景。本质上讲，一个元类会在定义时被触发，这时候你可以执行一些额外的操作。

下面是一个例子，利用这个思路来创建类似于 `collections` 模块中的命名元组的类：


```

import operator

class StructTupleMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for n, name in enumerate(cls._fields):
            setattr(cls, name, property(operator.itemgetter(n)))

class StructTuple(tuple, metaclass=StructTupleMeta):
    _fields = []
    def __new__(cls, *args):
        if len(args) != len(cls._fields):
            raise ValueError('{} arguments required'.format(len(cls._fields)))
        return super().__new__(cls, args)

```

这段代码可以用来定义简单的基于元组的数据结构，如下所示：

```

class Stock(StructTuple):
    _fields = ['name', 'shares', 'price']

class Point(StructTuple):
    _fields = ['x', 'y']

```

下面演示它如何工作：

```

>>> s = Stock('ACME', 50, 91.1)
>>> s
('ACME', 50, 91.1)
>>> s[0]
'ACME'
>>> s.name
'ACME'
>>> s.shares * s.price
4555.0
>>> s.shares = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>

```

讨论

这一小节中，类 `StructTupleMeta` 获取到类属性 `_fields` 中的属性名字列表，然后将它们转换成相应的可访问特定元组槽的方法。函数 `operator.itemgetter()` 创建一个访问器函数，然后 `property()` 函数将其转换成一个属性。

本节最难懂的部分是知道不同的初始化步骤是什么时候发生的。`StructTupleMeta` 中的 `__init__()` 方法只在每个类被定义时被调用一次。`cls` 参数就是那个被定义的类。实际上，上述代码使用了 `_fields` 类变量来保存新的被定义的类，然后给它再添加一点新的东西。

`StructTuple` 类作为一个普通的基类，供其他使用者来继承。这个类中的 `__new__()` 方法用来构造新的实例。这里使用 `__new__()` 并不是很常见，主要是因为我们要修改元组的调用签名，使得我们可以像普通的实例调用那样创建实例。就像下面这样：

```
s = Stock('ACME', 50, 91.1) # OK
s = Stock(('ACME', 50, 91.1)) # Error
```

跟 `__init__()` 不同的是，`__new__()` 方法在实例被创建之前被触发。由于元组是不可修改的，所以一旦它们被创建了就不可能对它做任何改变。而 `__init__()` 会在实例创建的最后被触发，这样的话我们就可以做我们想做的了。这也是为什么 `__new__()` 方法已经被定义了。

尽管本节很短，还是需要你能仔细研读，深入思考Python类是如何被定义的，实例是如何被创建的，还有就是元类和类的各个不同的方法究竟在什么时候被调用。

[PEP 422](#) 提供了一个解决本节问题的另外一种方法。但是，截止到我写这本书的时候，它还没被采纳和接受。尽管如此，如果你使用的是Python 3.3或更高的版本，那么还是值得去看一下的。

9.20 利用函数注解实现方法重载

问题

你已经学过怎样使用函数参数注解，那么你可能会想利用它来实现基于类型的方法重载。但是你不确定应该怎样去实现（或者到底行得通不）。

解决方案

本小节的技术是基于一个简单的技术，那就是Python允许参数注解，代码可以像下面这样写：

```

class Spam:
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)

    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

s = Spam()
s.bar(2, 3) # Prints Bar 1: 2 3
s.bar('hello') # Prints Bar 2: hello 0

```

下面是我们第一步的尝试，使用到了一个元类和描述器：

```

# multiple.py
import inspect
import types

class MultiMethod:
    """
    Represents a single multimethod.
    """
    def __init__(self, name):
        self._methods = {}
        self.__name__ = name

    def register(self, meth):
        """
        Register a new method as a multimethod
        """
        sig = inspect.signature(meth)

        # Build a type signature from the method's annotations
        types = []
        for name, parm in sig.parameters.items():
            if name == 'self':
                continue
            if parm.annotation is inspect.Parameter.empty:
                raise TypeError(
                    'Argument {} must be annotated with a type'.format(name)
                )
            if not isinstance(parm.annotation, type):
                raise TypeError(
                    'Argument {} annotation must be a type'.format(name)
                )
            if parm.default is not inspect.Parameter.empty:
                self._methods[tuple(types)] = meth
                types.append(parm.annotation)

        self._methods[tuple(types)] = meth

    def __call__(self, *args):
        """
        Call a method based on type signature of the arguments

```

```

    ...
    types = tuple(type(arg) for arg in args[1:])
    meth = self._methods.get(types, None)
    if meth:
        return meth(*args)
    else:
        raise TypeError('No matching method for types {}'.format(types))

def __get__(self, instance, cls):
    ...
    Descriptor method needed to make calls work in a class
    ...
    if instance is not None:
        return types.MethodType(self, instance)
    else:
        return self

class MultiDict(dict):
    ...
    Special dictionary to build multimethods in a metaclass
    ...
    def __setitem__(self, key, value):
        if key in self:
            # If key already exists, it must be a multimethod or callable
            current_value = self[key]
            if isinstance(current_value, MultiMethod):
                current_value.register(value)
            else:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else:
            super().__setitem__(key, value)

class MultipleMeta(type):
    ...
    Metaclass that allows multiple dispatch of methods
    ...
    def __new__(cls, clsname, bases, clsdict):
        return type.__new__(cls, clsname, bases, dict(clsdict))

    @classmethod
    def __prepare__(cls, clsname, bases):
        return MultiDict()

```

为了使用这个类，你可以像下面这样写：

```

class Spam(metaclass=MultipleMeta):
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)

    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

# Example: overloaded __init__
import time

class Date(metaclass=MultipleMeta):
    def __init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day

    def __init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)

```

下面是一个交互示例来验证它能正确的工作：

```

>>> s = Spam()
>>> s.bar(2, 3)
Bar 1: 2 3
>>> s.bar('hello')
Bar 2: hello 0
>>> s.bar('hello', 5)
Bar 2: hello 5
>>> s.bar(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 42, in __call__
    raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class 'int'>, <class 'str'>)
>>> # Overloaded __init__
>>> d = Date(2012, 12, 21)
>>> # Get today's date
>>> e = Date()
>>> e.year
2012
>>> e.month
12
>>> e.day
3
>>>

```

讨论

坦白来讲，相对于通常的代码而已本节使用到了很多的魔法代码。但是，它却能让我们深入理解元类和描述器的底层工作原理，并能加深对这些概念的印象。因此，就算你并不会立即去应用本节的技术，它的一些底层思想却会影响到其它涉及到元类、描述器和函数注解的编程技术。

本节的实现中的主要思路其实是很简单的。`MultipleMeta` 元类使用它的 `__prepare__()` 方法来提供一个作为 `MultiDict` 实例的自定义字典。这个跟普通字典不一样的是，`MultiDict` 会在元素被设置的时候检查是否已经存在，如果存在的话，重复的元素会在 `MultiMethod` 实例中合并。

`MultiMethod` 实例通过构建从类型签名到函数的映射来收集方法。在这个构建过程中，函数注解被用来收集这些签名然后构建这个映射。这个过程在 `MultiMethod.register()` 方法中实现。这种映射的一个关键特点是对于多个方法，所有参数类型都必须指定，否则就会报错。

为了让 `MultiMethod` 实例模拟一个调用，它的 `__call__()` 方法被实现了。这个方法从所有排除 `self` 的参数中构建一个类型元组，在内部 `map` 中查找这个方法，然后调用相应的方法。为了能让 `MultiMethod` 实例在类定义时正确操作，`__get__()` 是必须得实现的。它被用来构建正确的绑定方法。比如：

```
>>> b = s.bar
>>> b
<bound method Spam.bar of <__main__.Spam object at 0x1006a46d0>>
>>> b.__self__
<__main__.Spam object at 0x1006a46d0>
>>> b.__func__
<__main__.MultiMethod object at 0x1006a4d50>
>>> b(2, 3)
Bar 1: 2 3
>>> b('hello')
Bar 2: hello 0
>>>
```

不过本节的实现还有一些限制，其中一个它是不能使用关键字参数。例如：

```
>>> s.bar(x=2, y=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 'y'

>>> s.bar(s='hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 's'
>>>
```

也许有其他的方法能添加这种支持，但是它需要一个完全不同的方法映射方式。问题在于关键字参数的出现是没有顺序的。当它跟位置参数混合使用时，那你的参数就会变得比较混乱了，这时候你不得不在 `__call__()` 方法中先去做个排序。

同样对于继承也是有限制的，例如，类似下面这种代码就不能正常工作：

```
class A:
    pass

class B(A):
    pass

class C:
    pass

class Spam(metaclass=MultipleMeta):
    def foo(self, x:A):
        print('Foo 1:', x)

    def foo(self, x:C):
        print('Foo 2:', x)
```

原因是因为 `x:A` 注解不能成功匹配子类实例（比如B的实例），如下：

```
>>> s = Spam()
>>> a = A()
>>> s.foo(a)
Foo 1: <__main__.A object at 0x1006a5310>
>>> c = C()
>>> s.foo(c)
Foo 2: <__main__.C object at 0x1007a1910>
>>> b = B()
>>> s.foo(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 44, in __call__
    raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class '__main__.B'>,)
>>>
```

作为使用元类和注解的一种替代方案，可以通过描述器来实现类似的效果。例如：

```

import types

class multimethod:
    def __init__(self, func):
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def match(self, *types):
        def register(func):
            ndefaults = len(func.__defaults__) if func.__defaults__ else 0
            for n in range(ndefaults+1):
                self._methods[types[:len(types) - n]] = func
            return self
        return register

    def __call__(self, *args):
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            return self._default(*args)

    def __get__(self, instance, cls):
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self

```

为了使用描述器版本，你需要像下面这样写：

```

class Spam:
    @multimethod
    def bar(self, *args):
        # Default method called if no match
        raise TypeError('No matching method for bar')

    @bar.match(int, int)
    def bar(self, x, y):
        print('Bar 1:', x, y)

    @bar.match(str, int)
    def bar(self, s, n = 0):
        print('Bar 2:', s, n)

```

描述器方案同样也有前面提到的限制（不支持关键字参数和继承）。

所有事物都是平等的，有好有坏，也许最好的办法就是在普通代码中避免使用方法重载。不过有些特殊情况下还是有意义的，比如基于模式匹配的方法重载程序中。举个例子，8.21小节中的访问者模式可以修改为一个使用方法重载的类。但是，除了这个以外，通常不应该使用方法重载（就简单的使用不同名称的方法就行了）。

在Python社区对于实现方法重载的讨论已经由来已久。对于引发这个争论的原因，可以参考下Guido van Rossum的这篇博客：[Five-Minute Multimethods in Python](#)

9.21 避免重复的属性方法

问题

你在类中需要重复的定义一些执行相同逻辑的属性方法，比如进行类型检查，怎样去简化这些重复代码呢？

解决方案

考虑下一个简单的类，它的属性由属性方法包装：

```
class Person:
    def __init__(self, name ,age):
        self.name = name
        self.age = age

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('name must be a string')
        self._name = value

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if not isinstance(value, int):
            raise TypeError('age must be an int')
        self._age = value
```

可以看到，为了实现属性值的类型检查我们写了很多的重复代码。只要你以后看到类似这样的代码，你都应该想办法去简化它。一个可行的方法是创建一个函数用来定义属性并返回它。例如：

```

def typed_property(name, expected_type):
    storage_name = '_' + name

    @property
    def prop(self):
        return getattr(self, storage_name)

    @prop.setter
    def prop(self, value):
        if not isinstance(value, expected_type):
            raise TypeError('{} must be a {}'.format(name, expected_type))
        setattr(self, storage_name, value)

    return prop

# Example use
class Person:
    name = typed_property('name', str)
    age = typed_property('age', int)

    def __init__(self, name, age):
        self.name = name
        self.age = age

```

讨论

本节我们演示内部函数或者闭包的一个重要特性，它们很像一个宏。例子中的函数 `typed_property()` 看上去有点难理解，其实它所做的仅仅就是为你生成属性并返回这个属性对象。因此，当在一个类中使用它的时候，效果跟将它里面的代码放到类定义中去是一样的。尽管属性的 `getter` 和 `setter` 方法访问了本地变量如 `name`，`expected_type` 以及 `storage_name`，这个很正常，这些变量的值会保存在闭包当中。

我们还可以使用 `functools.partial()` 来稍稍改变下这个例子，很有趣。例如，你可以像下面这样：

```

from functools import partial

String = partial(typed_property, expected_type=str)
Integer = partial(typed_property, expected_type=int)

# Example:
class Person:
    name = String('name')
    age = Integer('age')

    def __init__(self, name, age):
        self.name = name
        self.age = age

```

其实你可以发现，这里的代码跟8.13小节中的类型系统描述器代码有些相似。

9.22 定义上下文管理器的简单方法

问题

你想自己去实现一个新的上下文管理器，以便使用with语句。

解决方案

实现一个新的上下文管理器的最简单的方法就是使用 `contextlib` 模块中的 `@contextmanager` 装饰器。下面是一个实现了代码块计时功能的上下文管理器例子：

```

import time
from contextlib import contextmanager

@contextmanager
def timethis(label):
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print('{}: {}'.format(label, end - start))

# Example use
with timethis('counting'):
    n = 10000000
    while n > 0:
        n -= 1

```

在函数 `timethis()` 中，`yield` 之前的代码会在上下文管理器中作为 `__enter__()` 方法执行，所有在 `yield` 之后的代码会作为 `__exit__()` 方法执行。如果出现了异常，异常会在 `yield` 语句那里抛出。

下面是一个更加高级一点的上下文管理器，实现了列表对象上的某种事务：

```
@contextmanager
def list_transaction(orig_list):
    working = list(orig_list)
    yield working
    orig_list[:] = working
```

这段代码的作用是任何对列表的修改只有当所有代码运行完成并且不出现异常的情况下才会生效。下面我们来演示一下：

```
>>> items = [1, 2, 3]
>>> with list_transaction(items) as working:
...     working.append(4)
...     working.append(5)
...
>>> items
[1, 2, 3, 4, 5]
>>> with list_transaction(items) as working:
...     working.append(6)
...     working.append(7)
...     raise RuntimeError('oops')
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: oops
>>> items
[1, 2, 3, 4, 5]
>>>
```

讨论

通常情况下，如果要写一个上下文管理器，你需要定义一个类，里面包含一个 `__enter__()` 和一个 `__exit__()` 方法，如下所示：

```
import time

class timethis:
    def __init__(self, label):
        self.label = label

    def __enter__(self):
        self.start = time.time()

    def __exit__(self, exc_ty, exc_val, exc_tb):
        end = time.time()
        print('{}: {}'.format(self.label, end - self.start))
```

尽管这个也不难写，但是相比较写一个简单的使用 `@contextmanager` 注解的函数而言还是稍显乏味。

`@contextmanager` 应该仅仅用来写自包含的上下文管理函数。如果你有一些对象(比如一个文件、网络连接或锁)，需要支持 `with` 语句，那么你就需要单独实现 `__enter__()` 方法和 `__exit__()` 方法。

9.23 在局部变量域中执行代码

问题

你想在使用范围内执行某个代码片段，并且希望在执行后所有的结果都不可见。

解决方案

为了理解这个问题，先试试一个简单场景。首先，在全局命名空间内执行一个代码片段：

```
>>> a = 13
>>> exec('b = a + 1')
>>> print(b)
14
>>>
```

然后，再在一个函数中执行同样的代码：

```
>>> def test():
...     a = 13
...     exec('b = a + 1')
...     print(b)
...
>>> test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in test
NameError: global name 'b' is not defined
>>>
```

可以看出，最后抛出了一个NameError异常，就跟在 `exec()` 语句从没执行过一样。要是你想在后面的计算中使用到 `exec()` 执行结果的话就会有问题了。

为了修正这样的错误，你需要在调用 `exec()` 之前使用 `locals()` 函数来得到一个局部变量字典。之后你就能从局部字典中获取修改过后的变量值了。例如：

```
>>> def test():
...     a = 13
...     loc = locals()
...     exec('b = a + 1')
...     b = loc['b']
...     print(b)
...
>>> test()
14
>>>
```

讨论

实际上对于 `exec()` 的正确使用是比较难的。大多数情况下当你要考虑使用 `exec()` 的时候，还有另外更好的解决方案（比如装饰器、闭包、元类等等）。

然而，如果你仍然要使用 `exec()`，本节列出了一些如何正确使用它的方法。默认情况下，`exec()` 会在调用者局部和全局范围内执行代码。然而，在函数里面，传递给 `exec()` 的局部范围是拷贝实际局部变量组成的一个字典。因此，如果 `exec()` 如果执行了修改操作，这种修改后的结果对实际局部变量值是没有影响的。下面是另外一个演示它的例子：

```
>>> def test1():
...     x = 0
...     exec('x += 1')
...     print(x)
...
>>> test1()
0
>>>
```

上面代码里，当你调用 `locals()` 获取局部变量时，你获得的是传递给 `exec()` 的局部变量的一个拷贝。通过在代码执行后审查这个字典的值，那就能获取修改后的值了。下面是一个演示例子：

```
>>> def test2():
...     x = 0
...     loc = locals()
...     print('before:', loc)
...     exec('x += 1')
...     print('after:', loc)
...     print('x =', x)
...
>>> test2()
before: {'x': 0}
after: {'loc': {...}, 'x': 1}
x = 0
>>>
```

仔细观察最后一步的输出，除非你将 `loc` 中被修改后的值手动赋值给 `x`，否则 `x` 变量值是不会变的。

在使用 `locals()` 的时候，你需要注意操作顺序。每次它被调用的时候，`locals()` 会获取局部变量值中的值并覆盖字典中相应的变量。请注意观察下下面这个试验的输出结果：

```
>>> def test3():
...     x = 0
...     loc = locals()
...     print(loc)
...     exec('x += 1')
...     print(loc)
...     locals()
...     print(loc)
...
>>> test3()
{'x': 0}
{'loc': {...}, 'x': 1}
{'loc': {...}, 'x': 0}
>>>
```

注意最后一次调用 `locals()` 的时候x的值是如何被覆盖掉的。

作为 `locals()` 的一个替代方案，你可以使用你自己的字典，并将它传递给 `exec()`。例如：

```
>>> def test4():
...     a = 13
...     loc = { 'a' : a }
...     glb = { }
...     exec('b = a + 1', glb, loc)
...     b = loc['b']
...     print(b)
...
>>> test4()
14
>>>
```

大部分情况下，这种方式是使用 `exec()` 的最佳实践。你只需要保证全局和局部字典在后面代码访问时已经被初始化。

还有一点，在使用 `exec()` 之前，你可能需要问下自己是否有其他更好的替代方案。大多数情况下当你要考虑使用 `exec()` 的时候，还有另外更好的解决方案，比如装饰器、闭包、元类，或其他一些元编程特性。

9.24 解析与分析Python源码

问题

你想写解析并分析Python源代码的程序。

解决方案

大部分程序员知道Python能够计算或执行字符串形式的源代码。例如：

```
>>> x = 42
>>> eval('2 + 3*4 + x')
56
>>> exec('for i in range(10): print(i)')
0
1
2
3
4
5
6
7
8
9
>>>
```

尽管如此，`ast` 模块能被用来将Python源码编译成一个可被分析的抽象语法树（AST）。例如：

```
>>> import ast
>>> ex = ast.parse('2 + 3*4 + x', mode='eval')
>>> ex
<_ast.Expression object at 0x1007473d0>
>>> ast.dump(ex)
"Expression(body=BinOp(left=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))), op=Add(),
right=Name(id='x', ctx=Load())))"

>>> top = ast.parse('for i in range(10): print(i)', mode='exec')
>>> top
<_ast.Module object at 0x100747390>
>>> ast.dump(top)
"Module(body=[For(target=Name(id='i', ctx=Store()),
iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)],
keywords=[], starargs=None, kwargs=None),
body=[Expr(value=Call(func=Name(id='print', ctx=Load()),
args=[Name(id='i', ctx=Load())], keywords=[], starargs=None,
kwargs=None)], or_else=[])])"
>>>
```

分析源码树需要你更多学习，它是由一系列AST节点组成的。分析这些节点最简单的方法就是定义一个访问者类，实现很多 `visit_NodeName()` 方法，`NodeName()` 匹配那些你感兴趣的节点。下面是这样一个类，记录了哪些名字被加载、存储和删除的信息。

```
import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()
        self.deleted = set()

    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
        elif isinstance(node.ctx, ast.Del):
            self.deleted.add(node.id)

# Sample usage
if __name__ == '__main__':
    # Some Python code
    code = '''
    for i in range(10):
        print(i)
    del i
    '''

    # Parse into an AST
    top = ast.parse(code, mode='exec')

    # Feed the AST to analyze name usage
    c = CodeAnalyzer()
    c.visit(top)
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)
```

如果你运行这个程序，你会得到下面这样的输出：

```
Loaded: {'i', 'range', 'print'}
Stored: {'i'}
Deleted: {'i'}
```

最后，AST可以通过 `compile()` 函数来编译并执行。例如：

```
>>> exec(compile(top, '<stdin>', 'exec'))
0
1
2
3
4
5
6
7
8
9
>>>
```

讨论

当你能够分析源代码并从中获取信息的时候，你就能写很多代码分析、优化或验证工具了。例如，相比盲目的传递一些代码片段到类似 `exec()` 函数中，你可以先将它转换成一个AST，然后观察它的细节看它到底是怎样做的。你还可以写一些工具来查看某个模块的全部源码，并且在此基础上执行某些静态分析。

需要注意的是，如果你知道自己在干啥，你还能够重写AST来表示新的代码。下面是一个装饰器例子，可以通过重新解析函数体源码、重写AST并重新创建函数代码对象来将全局访问变量降为函数体作用范围，

```

# nameLower.py
import ast
import inspect

# Node visitor that lowers globally accessed names into
# the function body as local variables.
class NameLower(ast.NodeVisitor):
    def __init__(self, lowered_names):
        self.lowered_names = lowered_names

    def visit_FunctionDef(self, node):
        # Compile some assignments to lower the constants
        code = '__globals = globals()\n'
        code += '\n'.join("{} = __globals['{}']".format(name)
                           for name in self.lowered_names)
        code_ast = ast.parse(code, mode='exec')

        # Inject new statements into the function body
        node.body[:0] = code_ast.body

        # Save the function object
        self.func = node

# Decorator that turns global names into locals
def lower_names(*namelist):
    def lower(func):
        srclines = inspect.getsource(func).splitlines()
        # Skip source lines prior to the @lower_names decorator
        for n, line in enumerate(srclines):
            if '@lower_names' in line:
                break

        src = '\n'.join(srclines[n+1:])
        # Hack to deal with indented code
        if src.startswith((' ', '\t')):
            src = 'if 1:\n' + src
        top = ast.parse(src, mode='exec')

        # Transform the AST
        cl = NameLower(namelist)
        cl.visit(top)

        # Execute the modified AST
        temp = {}
        exec(compile(top, '', 'exec'), temp, temp)

        # Pull out the modified code object
        func.__code__ = temp[func.__name__].__code__
        return func
    return lower

```

为了使用这个代码，你可以像下面这样写：

```
INCR = 1
@lower_names('INCR')
def countdown(n):
    while n > 0:
        n -= INCR
```

装饰器会将 `countdown()` 函数重写为类似下面这样子：

```
def countdown(n):
    __globals = globals()
    INCR = __globals['INCR']
    while n > 0:
        n -= INCR
```

在性能测试中，它会让函数运行快20%

现在，你是不是想为你所有的函数都加上这个装饰器呢？或许不会。但是，这却是对于一些高级技术比如AST操作、源码操作等等的一个很好的演示说明

本节受另外一个在 `ActiveState` 中处理Python字节码的章节的启示。使用AST是一个更加高级点的技术，并且也更简单些。参考下面一节获得字节码的更多信息。

9.25 拆解Python字节码

问题

你想通过将你的代码反编译成低级的字节码来查看它底层的工作机制。

解决方案

`dis` 模块可以被用来输出任何Python函数的反编译结果。例如：

```
>>> def countdown(n):
...     while n > 0:
...         print('T-minus', n)
...         n -= 1
...     print('Blastoff!')
...
>>> import dis
>>> dis.dis(countdown)
...
>>>
```

讨论

当你想要知道你的程序底层的运行机制的时候，`dis` 模块是很有用的。比如如果你想试着理解性能特征。被 `dis()` 函数解析的原始字节码如下所示：

```
>>> countdown.__code__.co_code
b"x'\x00|\x00\x00d\x01\x00k\x04\x00r)\x00t\x00\x00d\x02\x00|\x00\x00\x83
\x02\x00\x01|\x00\x00d\x03\x008}\x00\x00q\x03\x00Wt\x00\x00d\x04\x00\x83
\x01\x00\x01d\x00\x00S"
>>>
```

如果你想自己解释这段代码，你需要使用一些在 `opcode` 模块中定义的常量。例如：

```
>>> c = countdown.__code__.co_code
>>> import opcode
>>> opcode.opname[c[0]]
>>> opcode.opname[c[0]]
'SETUP_LOOP'
>>> opcode.opname[c[3]]
'LOAD_FAST'
>>>
```

奇怪的是，在 `dis` 模块中并没有函数让你以编程方式很容易的来处理字节码。不过，下面的生成器函数可以将原始字节码序列转换成 `opcodes` 和参数。

```

import opcode

def generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
    n = len(codebytes)
    while i < n:
        op = codebytes[i]
        i += 1
        if op >= opcode.HAVE_ARGUMENT:
            oparg = codebytes[i] + codebytes[i+1]*256 + extended_arg
            extended_arg = 0
            i += 2
            if op == opcode.EXTENDED_ARG:
                extended_arg = oparg * 65536
                continue
        else:
            oparg = None
        yield (op, oparg)

```

使用方法如下:

```

>>> for op, oparg in generate_opcodes(countdown.__code__.co_code):
...     print(op, opcode.opname[op], oparg)

```

这种方式很少有人知道，你可以利用它替换任何你想要替换的函数的原始字节码。下面我们用一个示例来演示整个过程:

```

>>> def add(x, y):
...     return x + y
...
>>> c = add.__code__
>>> c
<code object add at 0x1007beed0, file "<stdin>", line 1>
>>> c.co_code
b'|\x00\x00|\x01\x00\x17S'
>>>
>>> # Make a completely new code object with bogus byte code
>>> import types
>>> newbytecode = b'xxxxxxx'
>>> nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
...     c.co_nlocals, c.co_stacksize, c.co_flags, newbytecode, c.co_consts,
...     c.co_names, c.co_varnames, c.co_filename, c.co_name,
...     c.co_firstlineno, c.co_lnotab)
>>> nc
<code object add at 0x10069fe40, file "<stdin>", line 1>
>>> add.__code__ = nc
>>> add(2,3)
Segmentation fault

```

你可以像这样耍大招让解释器奔溃。但是，对于编写更高级优化和元编程工具的程序员来讲，他们可能真的需要重写字节码。本节最后的部分演示了这个是怎样做到的。你还可以参考另外一个类似的例子：[this code on ActiveState](#)

第十章：模块与包

模块与包是任何大型程序的核心，就连Python安装程序本身也是一个包。本章重点涉及有关模块和包的常用编程技术，例如如何组织包、把大型模块分割成多个文件、创建命名空间包。同时，也给出了让你自定义导入语句的秘籍。

Contents:

10.1 构建一个模块的层级包

问题

你想将你的代码组织成由很多分层模块构成的包。

解决方案

封装成包是很简单的。在文件系统上组织你的代码，并确保每个目录都定义了一个 `__init__.py` 文件。例如：

```
graphics/  
  __init__.py  
  primitive/  
    __init__.py  
    line.py  
    fill.py  
    text.py  
  formats/  
    __init__.py  
    png.py  
    jpg.py
```

一旦你做到了这一点，你应该能够执行各种 `import` 语句，如下：

```
import graphics.primitive.line  
from graphics.primitive import line  
import graphics.formats.jpg as jpg
```


讨论

定义模块的层次结构就像在文件系统上建立目录结构一样容易。文件`__init__.py`的目的是要包含不同运行级别的包的可选的初始化代码。举个例子，如果你执行了语句`import graphics`，文件`graphics/__init__.py`将被导入，建立`graphics`命名空间的内容。像`import graphics.format.jpg`这样导入，文件`graphics/__init__.py`和文件`graphics/graphics/formats/__init__.py`将在文件`graphics/formats/jpg.py`导入之前导入。

绝大部分时候让`__init__.py`空着就好。但是有些情况下可能包含代码。举个例子，`__init__.py`能够用来自动加载子模块：

```
# graphics/formats/__init__.py
from . import jpg
from . import png
```

像这样一个文件，用户可以仅仅通过`import graphics.formats`来代替`import graphics.formats.jpg`以及`import graphics.formats.png`。

`__init__.py`的其他常用用法包括将多个文件合并到一个逻辑命名空间，这将在10.4小节讨论。

敏锐的程序员会发现，即使没有`__init__.py`文件存在，python仍然会导入包。如果你没有定义`__init__.py`时，实际上创建了一个所谓的“命名空间包”，这将在10.5小节讨论。万物平等，如果你着手创建一个新的包的话，包含一个`__init__.py`文件吧。

10.2 控制模块被全部导入的内容

问题

当使用`'from module import *`语句时，希望对从模块或包导出的符号进行精确控制。

解决方案

在你的模块中定义一个变量`__all__`来明确地列出需要导出的内容。

举个例子：

```
# somemodule.py
def spam():
    pass

def grok():
    pass

blah = 42
# Only export 'spam' and 'grok'
__all__ = ['spam', 'grok']
```

讨论

尽管强烈反对使用‘`from module import *`’,但是在定义了大量变量名的模块中频繁使用。如果你不做任何事,这样的导入将会导入所有不以下划线开头的。另一方面,如果定义了`__all__`,那么只有被列举出的东西会被导出。

如果你将`__all__`定义成一个空列表,没有东西将被导出。如果`__all__`包含未定义的名字,在导入时引起`AttributeError`。

10.3 使用相对路径名导入包中子模块

问题

将代码组织成包,想用`import`语句从另一个包名没有硬编码过的包的中导入子模块。

解决方案

使用包的相对导入,使一个的模块导入同一个包的另一个模块 举个例子,假设在你的文件系统上有`mypackage`包,组织如下:

```
mypackage/
  __init__.py
  A/
    __init__.py
    spam.py
    grok.py
  B/
    __init__.py
    bar.py
```

如果模块`mypackage.A.spam`要导入同目录下的模块`grok`,它应该包括的`import`语句如下:

```
# mypackage/A/spam.py
from . import grok
```

如果模块mypackage.A.spam要导入不同目录下的模块B.bar，它应该使用的import语句如下：

```
# mypackage/A/spam.py
from ..B import bar
```

两个import语句都没包含顶层包名，而是使用了spam.py的相对路径。

讨论

在包内，既可以使用相对路径也可以使用绝对路径来导入。举个例子：

```
# mypackage/A/spam.py
from mypackage.A import grok # OK
from . import grok # OK
import grok # Error (not found)
```

像mypackage.A这样使用绝对路径名的不利之处是这将顶层包名硬编码到你的源码中。如果你想重新组织它，你的代码将更脆，很难工作。举个例子，如果你改变了包名，你就必须检查所有文件来修正源码。同样，硬编码的名称会使移动代码变得困难。举个例子，也许有人想安装两个不同版本的软件包，只通过名称区分它们。如果使用相对导入，那一切都ok，然而使用绝对路径名很可能会出问题。

import语句的`.`和`..`看起来很滑稽,但它指定目录名为当前目录，`..B`为目录`../B`。这种语法只适用于import。举个例子：

```
from . import grok # OK
import .grok # ERROR
```

尽管使用相对导入看起来像是浏览文件系统，但是不能到定义包的目录之外。也就是说，使用点的这种模式从不是包的目录中导入将会引发错误。

最后，相对导入只适用于在合适的包中的模块。尤其是在顶层的脚本的简单模块中，它们将不起作用。如果包的部分被作为脚本直接执行，那它们将不起作用 例如：

```
% python3 mypackage/A/spam.py # Relative imports fail
```

另一方面，如果你使用Python的-m选项来执行先前的脚本，相对导入将会正确运行。例如：

```
% python3 -m mypackage.A.spam # Relative imports work
```

更多的包的相对导入的背景知识,请看 [PEP 328](#) .

10.4 将模块分割成多个文件

问题

你想将一个模块分割成多个文件。但是你不将分离的文件统一成一个逻辑模块时使已有的代码遭到破坏。

解决方案

程序模块可以通过变成包来分割成多个独立的文件。考虑下下面简单的模块：

```
# mymodule.py
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def bar(self):
        print('B.bar')
```

假设你想mymodule.py分为两个文件，每个定义的一个类。要做到这一点，首先用mymodule目录来替换文件mymodule.py。在这个目录下，创建以下文件：

```
mymodule/  
  __init__.py  
  a.py  
  b.py
```

在a.py文件中插入以下代码：

```
# a.py  
class A:  
    def spam(self):  
        print('A.spam')
```

在b.py文件中插入以下代码：

```
# b.py  
from .a import A  
class B(A):  
    def bar(self):  
        print('B.bar')
```

最后，在__init__.py中，将2个文件粘合在一起：

```
# __init__.py  
from .a import A  
from .b import B
```

如果按照这些步骤，所产生的包MyModule将作为一个单一的逻辑模块：

```
>>> import mymodule  
>>> a = mymodule.A()  
>>> a.spam()  
A.spam  
>>> b = mymodule.B()  
>>> b.bar()  
B.bar  
>>>
```

讨论

在这个章节中的主要问题是一个设计问题，不管你是否希望用户使用很多小模块或只是一个模块。举个例子，在一个大型的代码库中，你可以将这一切都分割成独立的文件，让用户使用大量的import语句，就像这样：

```
from mymodule.a import A
from mymodule.b import B
...
```

这样能工作，但这让用户承受更多的负担，用户要知道不同的部分位于何处。通常情况下，将这些统一起来，使用一条import将更加容易，就像这样：

```
from mymodule import A, B
```

对后者而言，让mymodule成为一个大的源文件是最常见的。但是，这一章节展示了如何合并多个文件合并成一个单一的逻辑命名空间。这样做的关键是创建一个包目录，使用__init__.py文件来将每部分粘合在一起。

当一个模块被分割，你需要特别注意交叉引用的文件名。举个例子，在这一章节中，B类需要访问A类作为基类。用包的相对导入from .a import A来获取。

整个章节都使用包的相对导入来避免将顶层模块名硬编码到源代码中。这使得重命名模块或者将它移动到别的位置更容易。（见10.3小节）

作为这一章节的延伸，将介绍延迟导入。如图所示，__init__.py文件一次导入所有必需的组件的。但是对于一个很大的模块，可能你只想组件在需要时被加载。要做到这一点，__init__.py有细微的变化：

```
# __init__.py
def A():
    from .a import A
    return A()

def B():
    from .b import B
    return B()
```

在这个版本中，类A和类B被替换为在第一次访问时加载所需的类的函数。对于用户，这看起来不会有太大的不同。例如：

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>>
```

延迟加载的主要缺点是继承和类型检查可能会中断。你可能会稍微改变你的代码，例如：

```
if isinstance(x, mymodule.A): # Error
...

if isinstance(x, mymodule.a.A): # Ok
...
```

延迟加载的真实例子, 见标准库 `multiprocessing/_init_.py` 的源码.

10.5 利用命名空间导入目录分散的代码

问题

你可能有大量的代码，由不同的人来分散地维护。每个部分被组织为文件目录，如一个包。然而，你希望能用共同的包前缀将所有组件连接起来，不是将每一个部分作为独立的包来安装。

解决方案

从本质上讲，你要定义一个顶级Python包，作为一个大集合分开维护子包的命名空间。这个问题经常出现在大的应用框架中，框架开发者希望鼓励用户发布插件或附加包。

在统一不同的目录里统一相同的命名空间，但是要删去用来将组件联合起来的`_init_.py`文件。假设你有Python代码的两个不同的目录如下：

```
foo-package/
  spam/
    blah.py

bar-package/
  spam/
    grok.py
```

在这2个目录里，都有着共同的命名空间spam。在任何一个目录里都没有__init__.py文件。

让我们看看，如果将foo-package和bar-package都加到python模块路径并尝试导入会发生什么

```
>>> import sys
>>> sys.path.extend(['foo-package', 'bar-package'])
>>> import spam.blah
>>> import spam.grok
>>>
```

两个不同的包目录被合并到一起，你可以导入spam.blah和spam.grok，并且它们能够工作。

讨论

在这里工作的机制被称为“包命名空间”的一个特征。从本质上讲，包命名空间是一种特殊的封装设计，为合并不同的目录的代码到一个共同的命名空间。对于大的框架，这可能是有用的，因为它允许一个框架的部分被单独地安装下载。它也使人们能够轻松地这样的框架编写第三方附加组件和其他扩展。

包命名空间的关键是确保顶级目录中没有__init__.py文件来作为共同的命名空间。缺失__init__.py文件使得在导入包的时候会发生有趣的事情：这并没有产生错误，解释器创建了一个由所有包含匹配包名的目录组成的列表。特殊的包命名空间模块被创建，只读的目录列表副本被存储在其__path__变量中。举个例子：

```
>>> import spam
>>> spam.__path__
_NamespacePath(['foo-package/spam', 'bar-package/spam'])
>>>
```

在定位包的子组件时，目录__path__将被用到(例如,当导入spam.grok或者spam.blah的时候).

包命名空间的一个重要特点是任何人都可以用自己的代码来扩展命名空间。举个例子，假设你自己的代码目录像这样：


```
my-package/  
  spam/  
    custom.py
```

如果你将你的代码目录和其他包一起添加到`sys.path`，这将无缝地合并到别的`spam`包目录中：

```
>>> import spam.custom  
>>> import spam.grok  
>>> import spam.blah  
>>>
```

一个包是否被作为一个包命名空间的主要方法是检查其`__file__`属性。如果没有，那包是个命名空间。这也可以由其字符表现形式中的“namespace”这个词体现出来。

```
>>> spam.__file__  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'module' object has no attribute '__file__'  
>>> spam  
<module 'spam' (namespace)>  
>>>
```

更多的包命名空间信息可以查看 [PEP 420](#)。

10.6 重新加载模块

问题

你想重新加载已经加载的模块，因为你对其源码进行了修改。

解决方案

使用`imp.reload()`来重新加载先前加载的模块。举个例子：

```
>>> import spam
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>>
```

讨论

重新加载模块在开发和调试过程中常常很有用。但在生产环境中的代码使用会不安全，因为它并不总是像您期望的那样工作。

`reload()`擦除了模块底层字典的内容，并通过重新执行模块的源代码来刷新它。模块对象本身的身份保持不变。因此，该操作在程序中所有已经被导入了的地方更新了模块。

尽管如此，`reload()`没有更新像“`from module import name`”这样使用`import`语句导入的定义。举个例子：

```
# spam.py
def bar():
    print('bar')

def grok():
    print('grok')
```

现在启动交互式会话：

```
>>> import spam
>>> from spam import grok
>>> spam.bar()
bar
>>> grok()
grok
>>>
```

不退出Python修改`spam.py`的源码，将`grok()`函数改成这样：

```
def grok():
    print('New grok')
```

现在回到交互式会话，重新加载模块，尝试下这个实验：

```
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>> spam.bar()
bar
>>> grok() # Notice old output
grok
>>> spam.grok() # Notice new output
New grok
>>>
```

在这个例子中，你看到有2个版本的`grok()`函数被加载。通常来说，这不是你想要的，而是令人头疼的事。

因此，在生产环境中可能需要避免重新加载模块。在交互环境下调试，解释程序并试图弄懂它。

10.7 运行目录或压缩文件

问题

您有已经一个复杂的脚本到涉及多个文件的应用程序。你想有一些简单的方法让用户运行程序。

解决方案

如果你的应用程序已经有多个文件，你可以把你的应用程序放进它自己的目录并添加一个`__main__.py`文件。举个例子，你可以像这样创建目录：

```
myapplication/
  spam.py
  bar.py
  grok.py
  __main__.py
```

如果`__main__.py`存在，你可以简单地在顶级目录运行Python解释器：

```
bash % python3 myapplication
```

解释器将执行__main__.py文件作为主程序。

如果你将你的代码打包成zip文件，这种技术同样也适用，举个例子：

```
bash % ls
spam.py bar.py grok.py __main__.py
bash % zip -r myapp.zip *.py
bash % python3 myapp.zip
... output from __main__.py ...
```

讨论

创建一个目录或zip文件并添加__main__.py文件来将一个更大的Python应用打包是可行的。这和作为标准库被安装到Python库的代码包是有一点区别的。相反，这只是让别人执行的代码包。

由于目录和zip文件与正常文件有一点不同，你可能还需要增加一个shell脚本，使执行更加容易。例如，如果代码文件名为myapp.zip，你可以创建这样一个顶级脚本：

```
#!/usr/bin/env python3 /usr/local/bin/myapp.zip
```

10.8 读取位于包中的数据文件

问题

你的包中包含代码需要去读取的数据文件。你需要尽可能地用最便捷的方式来做这件事。

解决方案

假设你的包中的文件组织成如下：

```
mypackage/
  __init__.py
  somedata.dat
  spam.py
```

现在假设spam.py文件需要读取somedata.dat文件中的内容。你可以用以下代码来完成：

```
# spam.py
import pkgutil
data = pkgutil.get_data(__package__, 'somedata.dat')
```

由此产生的变量是包含该文件的原始内容的字节字符串。

讨论

要读取数据文件，你可能会倾向于编写使用内置的I/O功能的代码，如open()。但是这种方法也有一些问题。

首先，一个包对解释器的当前工作目录几乎没有控制权。因此，编程时任何I/O操作都必须使用绝对文件名。由于每个模块包含有完整路径的`_file_`变量，这弄清楚它的路径不是不可能，但它很凌乱。

第二，包通常安装作为zip或egg文件，这些文件像文件系统上的一个普通目录一样不会被保留。因此，你试图用open()对一个包含数据文件的归档文件进行操作，它根本不会工作。

pkgutil.get_data()函数是一个读取数据文件的高级工具，不用管包是如何安装以及安装在哪儿。它只是工作并将文件内容以字节字符串返回给你

get_data()的第一个参数是包含包名的字符串。你可以直接使用包名，也可以使用特殊的变量，比如`__package__`。第二个参数是包内文件的相对名称。如果有必要，可以使用标准的Unix命名规范到不同的目录，只有最后的目录仍然位于包中。

10.9 将文件夹加入到sys.path

问题

你无法导入你的Python代码因为它所在的目录不在sys.path里。你想将添加新目录到Python路径，但是不想硬链接到你的代码。

解决方案

有两种常用的方式将新目录添加到`sys.path`。第一种，你可以使用`PYTHONPATH`环境变量来添加。例如：

```
bash % env PYTHONPATH=/some/dir:/other/dir python3
Python 3.3.0 (default, Oct 4 2012, 10:17:33)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/some/dir', '/other/dir', ...]
>>>
```

在自定义应用程序中，这样的环境变量可在程序启动时设置或通过`shell`脚本。

第二种方法是创建一个`.pth`文件，将目录列举出来，像这样：

```
# myapplication.pth
/some/dir
/other/dir
```

这个`.pth`文件需要放在某个Python的`site-packages`目录，通常位于`/usr/local/lib/python3.3/site-packages` 或者 `~/local/lib/python3.3/sitepackages`。当解释器启动时，`.pth`文件里列举出来的存在于文件系统的目录将被添加到`sys.path`。安装一个`.pth`文件可能需要管理员权限，如果它被添加到系统级的Python解释器。

讨论

比起费力地找文件，你可能会倾向于写一个代码手动调节`sys.path`的值。例如：

```
import sys
sys.path.insert(0, '/some/dir')
sys.path.insert(0, '/other/dir')
```

虽然这能“工作”，它是在实践中极为脆弱，应尽量避免使用。这种方法的问题是，它将目录名硬编码到了你的源。如果你的代码被移到一个新的位置，这会导致维护问题。更好的做法是在不修改源代码的情况下，将`path`配置到其他地方。如果您使用模块级的变量来精心构造一个适当的绝对路径，有时你可以解决硬编码目录的问题，比如`_file_`。举个例子：

```
import sys
from os.path import abspath, join, dirname
sys.path.insert(0, abspath(dirname('__file__'), 'src'))
```

这将src目录添加到path里，和执行插入步骤的代码在同一个目录里。

site-packages目录是第三方包和模块安装的目录。如果你手动安装你的代码，它将被安装到site-packages目录。虽然.pth文件配置的path必须出现在site-packages里，但代码可以在系统上任何你想要的目录。因此，你可以把你的代码放在一系列不同的目录，只要那些目录包含在.pth文件里。

10.10 通过字符串名导入模块

问题

你想导入一个模块，但是模块的名字在字符串里。你想对字符串调用导入命令。

解决方案

使用importlib.import_module()函数来手动导入名字为字符串给出的一个模块或者包的一部分。举个例子：

```
>>> import importlib
>>> math = importlib.import_module('math')
>>> math.sin(2)
0.9092974268256817
>>> mod = importlib.import_module('urllib.request')
>>> u = mod.urlopen('http://www.python.org')
>>>
```

import_module只是简单地执行和import相同的步骤，但是返回生成的模块对象。你只需要将其存储在一个变量，然后像正常的模块一样使用。

如果你正在使用的包，import_module()也可用于相对导入。但是，你需要给它一个额外的参数。例如：

```
import importlib
# Same as 'from . import b'
b = importlib.import_module('.b', __package__)
```

讨论

使用`import_module()`手动导入模块的问题通常出现在以某种方式编写修改或覆盖模块的代码时候。例如，也许你正在执行某种自定义导入机制，需要通过名称来加载一个模块，通过补丁加载代码。

在旧的代码，有时你会看到用于导入的内建函数`_import_()`。尽管它能工作，但是`importlib.import_module()`通常更容易使用。

自定义导入过程的高级实例见10.11小节

10.11 通过导入钩子远程加载模块

问题

You would like to customize Python's import statement so that it can transparently load modules from a remote machine.

解决方案

First, a serious disclaimer about security. The idea discussed in this recipe would be wholly bad without some kind of extra security and authentication layer. That said, the main goal is actually to take a deep dive into the inner workings of Python's import statement. If you get this recipe to work and understand the inner workings, you'll have a solid foundation of customizing import for almost any other purpose. With that out of the way, let's carry on.

At the core of this recipe is a desire to extend the functionality of the import statement. There are several approaches for doing this, but for the purposes of illustration, start by making the following directory of Python code:

```
testcode/  
  spam.py  
  fib.py  
  grok/  
    __init__.py  
    blah.py
```

The content of these files doesn't matter, but put a few simple statements and functions in each file so you can test them and see output when they're imported. For example:


```

# spam.py
print("I'm spam")

def hello(name):
    print('Hello %s' % name)

# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

# grok/__init__.py
print("I'm grok.__init__")

# grok/blah.py
print("I'm grok.blah")

```

The goal here is to allow remote access to these files as modules. Perhaps the easiest way to do this is to publish them on a web server. Simply go to the testcode directory and run Python like this:

```

bash % cd testcode
bash % python3 -m http.server 15000
Serving HTTP on 0.0.0.0 port 15000 ...

```

Leave that server running and start up a separate Python interpreter. Make sure you can access the remote files using urllib. For example:

```

>>> from urllib.request import urlopen
>>> u = urlopen('http://localhost:15000/fib.py')
>>> data = u.read().decode('utf-8')
>>> print(data)
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>

```

Loading source code from this server is going to form the basis for the remainder of this recipe. Specifically, instead of manually grabbing a file of source code using `urlopen()`, the `import` statement will be customized to do it transparently behind the scenes.

The first approach to loading a remote module is to create an explicit loading function for doing it. For example:

```
import imp
import urllib.request
import sys

def load_module(url):
    u = urllib.request.urlopen(url)
    source = u.read().decode('utf-8')
    mod = sys.modules.setdefault(url, imp.new_module(url))
    code = compile(source, url, 'exec')
    mod.__file__ = url
    mod.__package__ = ''
    exec(code, mod.__dict__)
    return mod
```

This function merely downloads the source code, compiles it into a code object using `compile()`, and executes it in the dictionary of a newly created module object. Here's how you would use the function:

```
>>> fib = load_module('http://localhost:15000/fib.py')
I'm fib
>>> fib.fib(10)
89
>>> spam = load_module('http://localhost:15000/spam.py')
I'm spam
>>> spam.hello('Guido')
Hello Guido
>>> fib
<module 'http://localhost:15000/fib.py' from 'http://localhost:15000/fib.py'>
>>> spam
<module 'http://localhost:15000/spam.py' from 'http://localhost:15000/spam.py'>
>>>
```

As you can see, it “works” for simple modules. However, it’s not plugged into the usual `import` statement, and extending the code to support more advanced constructs, such as packages, would require additional work.

A much slicker approach is to create a custom importer. The first way to do this is to create what’s known as a meta path importer. Here is an example:

```

# urlimport.py
import sys
import importlib.abc
import imp
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
from html.parser import HTMLParser

# Debugging
import logging
log = logging.getLogger(__name__)

# Get links from a given URL
def _get_links(url):
    class LinkParser(HTMLParser):
        def handle_starttag(self, tag, attrs):
            if tag == 'a':
                attrs = dict(attrs)
                links.add(attrs.get('href').rstrip('/'))
    links = set()
    try:
        log.debug('Getting links from %s' % url)
        u = urlopen(url)
        parser = LinkParser()
        parser.feed(u.read().decode('utf-8'))
    except Exception as e:
        log.debug('Could not get links. %s', e)
    log.debug('links: %r', links)
    return links

class UrlMetaFinder(importlib.abc.MetaPathFinder):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._links = { }
        self._loaders = { baseurl : UrlModuleLoader(baseurl) }

    def find_module(self, fullname, path=None):
        log.debug('find_module: fullname=%r, path=%r', fullname, path)
        if path is None:
            baseurl = self._baseurl
        else:
            if not path[0].startswith(self._baseurl):
                return None
            baseurl = path[0]
        parts = fullname.split('.')
        basename = parts[-1]
        log.debug('find_module: baseurl=%r, basename=%r', baseurl, basename)

        # Check Link cache
        if basename not in self._links:
            self._links[baseurl] = _get_links(baseurl)

        # Check if it's a package
        if basename in self._links[baseurl]:
            log.debug('find_module: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init__.py)

```

```

        loader = UrlPackageLoader(fullurl)
    try:
        loader.load_module(fullname)
        self._links[fullurl] = _get_links(fullurl)
        self._loaders[fullurl] = UrlModuleLoader(fullurl)
        log.debug('find_module: package %r loaded', fullname)
    except ImportError as e:
        log.debug('find_module: package failed. %s', e)
        loader = None
    return loader

# A normal module
filename = basename + '.py'
if filename in self._links[baseurl]:
    log.debug('find_module: module %r found', fullname)
    return self._loaders[baseurl]
else:
    log.debug('find_module: module %r not found', fullname)
    return None

def invalidate_caches(self):
    log.debug('invalidating link cache')
    self._links.clear()

# Module Loader for a URL
class UrlModuleLoader(importlib.abc.SourceLoader):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._source_cache = {}

    def module_repr(self, module):
        return '<urlmodule %r from %r>' % (module.__name__, module.__file__)

# Required method
def load_module(self, fullname):
    code = self.get_code(fullname)
    mod = sys.modules.setdefault(fullname, imp.new_module(fullname))
    mod.__file__ = self.get_filename(fullname)
    mod.__loader__ = self
    mod.__package__ = fullname.rpartition('.')[0]
    exec(code, mod.__dict__)
    return mod

# Optional extensions
def get_code(self, fullname):
    src = self.get_source(fullname)
    return compile(src, self.get_filename(fullname), 'exec')

def get_data(self, path):
    pass

def get_filename(self, fullname):
    return self._baseurl + '/' + fullname.split('.')[0] + '.py'

def get_source(self, fullname):
    filename = self.get_filename(fullname)
    log.debug('loader: reading %r', filename)
    if filename in self._source_cache:
        log.debug('loader: cached %r', filename)

```

```

        return self._source_cache[filename]
    try:
        u = urlopen(filename)
        source = u.read().decode('utf-8')
        log.debug('loader: %r loaded', filename)
        self._source_cache[filename] = source
        return source
    except (HTTPError, URLError) as e:
        log.debug('loader: %r failed. %s', filename, e)
        raise ImportError("Can't load %s" % filename)

def is_package(self, fullname):
    return False

# Package loader for a URL
class UrlPackageLoader(UrlModuleLoader):
    def load_module(self, fullname):
        mod = super().load_module(fullname)
        mod.__path__ = [ self._baseurl ]
        mod.__package__ = fullname

    def get_filename(self, fullname):
        return self._baseurl + '/' + '__init__.py'

    def is_package(self, fullname):
        return True

# Utility functions for installing/uninstalling the loader
_installed_meta_cache = { }
def install_meta(address):
    if address not in _installed_meta_cache:
        finder = UrlMetaFinder(address)
        _installed_meta_cache[address] = finder
        sys.meta_path.append(finder)
        log.debug('%r installed on sys.meta_path', finder)

def remove_meta(address):
    if address in _installed_meta_cache:
        finder = _installed_meta_cache.pop(address)
        sys.meta_path.remove(finder)
        log.debug('%r removed from sys.meta_path', finder)

```

Here is an interactive session showing how to use the preceding code:

```

>>> # importing currently fails
>>> import fib
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> # Load the importer and retry (it works)
>>> import urlimport
>>> urlimport.install_meta('http://localhost:15000')
>>> import fib
I'm fib
>>> import spam
I'm spam
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>

```

This particular solution involves installing an instance of a special finder object `UrlMetaFinder` as the last entry in `sys.meta_path`. Whenever modules are imported, the finders in `sys.meta_path` are consulted in order to locate the module. In this example, the `UrlMetaFinder` instance becomes a finder of last resort that's triggered when a module can't be found in any of the normal locations.

As for the general implementation approach, the `UrlMetaFinder` class wraps around a user-specified URL. Internally, the finder builds sets of valid links by scraping them from the given URL. When imports are made, the module name is compared against this set of known links. If a match can be found, a separate `UrlModuleLoader` class is used to load source code from the remote machine and create the resulting module object. One reason for caching the links is to avoid unnecessary HTTP requests on repeated imports.

The second approach to customizing import is to write a hook that plugs directly into the `sys.path` variable, recognizing certain directory naming patterns. Add the following class and support functions to `urlimport.py`:

```

# urlimport.py
# ... include previous code above ...
# Path finder class for a URL
class UrlPathFinder(importlib.abc.PathEntryFinder):
    def __init__(self, baseurl):
        self._links = None
        self._loader = UrlModuleLoader(baseurl)
        self._baseurl = baseurl

    def find_loader(self, fullname):
        log.debug('find_loader: %r', fullname)
        parts = fullname.split('.')

```

```

basename = parts[-1]
# Check link cache
if self._links is None:
    self._links = [] # See discussion
    self._links = _get_links(self._baseurl)

# Check if it's a package
if basename in self._links:
    log.debug('find_loader: trying package %r', fullname)
    fullurl = self._baseurl + '/' + basename
    # Attempt to load the package (which accesses __init__.py)
    loader = UrlPackageLoader(fullurl)
    try:
        loader.load_module(fullname)
        log.debug('find_loader: package %r loaded', fullname)
    except ImportError as e:
        log.debug('find_loader: %r is a namespace package', fullname)
        loader = None
    return (loader, [fullurl])

# A normal module
filename = basename + '.py'
if filename in self._links:
    log.debug('find_loader: module %r found', fullname)
    return (self._loader, [])
else:
    log.debug('find_loader: module %r not found', fullname)
    return (None, [])

def invalidate_caches(self):
    log.debug('invalidating link cache')
    self._links = None

# Check path to see if it looks like a URL
_url_path_cache = {}
def handle_url(path):
    if path.startswith(('http://', 'https://')):
        log.debug('Handle path? %s. [Yes]', path)
        if path in _url_path_cache:
            finder = _url_path_cache[path]
        else:
            finder = UrlPathFinder(path)
            _url_path_cache[path] = finder
        return finder
    else:
        log.debug('Handle path? %s. [No]', path)

def install_path_hook():
    sys.path_hooks.append(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Installing handle_url')

def remove_path_hook():
    sys.path_hooks.remove(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Removing handle_url')

```

To use this path-based finder, you simply add URLs to `sys.path`. For example:

```
>>> # Initial import fails
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Install the path hook
>>> import urlimport
>>> urlimport.install_path_hook()

>>> # Imports still fail (not on path)
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Add an entry to sys.path and watch it work
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
I'm fib
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

The key to this last example is the `handle_url()` function, which is added to the `sys.path_hooks` variable. When the entries on `sys.path` are being processed, the functions in `sys.path_hooks` are invoked. If any of those functions return a finder object, that finder is used to try to load modules for that entry on `sys.path`.

It should be noted that the remotely imported modules work exactly like any other module. For instance:


```
>>> fib
<urlmodule 'fib' from 'http://localhost:15000/fib.py'>
>>> fib.__name__
'fib'
>>> fib.__file__
'http://localhost:15000/fib.py'
>>> import inspect
>>> print(inspect.getsource(fib))
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>
```

讨论

Before discussing this recipe in further detail, it should be emphasized that Python's module, package, and import mechanism is one of the most complicated parts of the entire language—often poorly understood by even the most seasoned Python programmers unless they've devoted effort to peeling back the covers. There are several critical documents that are worth reading, including the documentation for the [importlib module](#) and [PEP 302](#). That documentation won't be repeated here, but some essential highlights will be discussed.

First, if you want to create a new module object, you use the `imp.new_module()` function. For example:

```
>>> import imp
>>> m = imp.new_module('spam')
>>> m
<module 'spam'>
>>> m.__name__
'spam'
>>>
```

Module objects usually have a few expected attributes, including `__file__` (the name of the file that the module was loaded from) and `__package__` (the name of the enclosing package, if any).

Second, modules are cached by the interpreter. The module cache can be found in the dictionary `sys.modules`. Because of this caching, it's common to combine caching and module creation together into a single step. For example:

```
>>> import sys
>>> import imp
>>> m = sys.modules.setdefault('spam', imp.new_module('spam'))
>>> m
<module 'spam'>
>>>
```

The main reason for doing this is that if a module with the given name already exists, you'll get the already created module instead. For example:

```
>>> import math
>>> m = sys.modules.setdefault('math', imp.new_module('math'))
>>> m
<module 'math' from '/usr/local/lib/python3.3/lib-dynload/math.so'>
>>> m.sin(2)
0.9092974268256817
>>> m.cos(2)
-0.4161468365471424
>>>
```

Since creating modules is easy, it is straightforward to write simple functions, such as the `load_module()` function in the first part of this recipe. A downside of this approach is that it is actually rather tricky to handle more complicated cases, such as package imports. In order to handle a package, you would have to reimplement much of the underlying logic that's already part of the normal import statement (e.g., checking for directories, looking for `__init__.py` files, executing those files, setting up paths, etc.). This complexity is one of the reasons why it's often better to extend the import statement directly rather than defining a custom function.

Extending the import statement is straightforward, but involves a number of moving parts. At the highest level, import operations are processed by a list of "meta-path" finders that you can find in the list `sys.meta_path`. If you output its value, you'll see the following:

```
>>> from pprint import pprint
>>> pprint(sys.meta_path)
[<class '_frozen_importlib.BuiltinImporter'>,
 <class '_frozen_importlib.FrozenImporter'>,
 <class '_frozen_importlib.PathFinder'>]
>>>
```

When executing a statement such as `import fib`, the interpreter walks through the finder objects on `sys.meta_path` and invokes their `find_module()` method in order to locate an appropriate module loader. It helps to see this by experimentation, so define the following class and try the following:

```
>>> class Finder:
...     def find_module(self, fullname, path):
...         print('Looking for', fullname, path)
...         return None
...
>>> import sys
>>> sys.meta_path.insert(0, Finder()) # Insert as first entry
>>> import math
Looking for math None
>>> import types
Looking for types None
>>> import threading
Looking for threading None
Looking for time None
Looking for traceback None
Looking for linecache None
Looking for tokenize None
Looking for token None
>>>
```

Notice how the `find_module()` method is being triggered on every import. The role of the path argument in this method is to handle packages. When packages are imported, it is a list of the directories that are found in the package's `__path__` attribute. These are the paths that need to be checked to find package subcomponents. For example, notice the path setting for `xml.etree` and `xml.etree.ElementTree`:

```
>>> import xml.etree.ElementTree
Looking for xml None
Looking for xml.etree ['/usr/local/lib/python3.3/xml']
Looking for xml.etree.ElementTree ['/usr/local/lib/python3.3/xml/etree']
Looking for warnings None
Looking for contextlib None
Looking for xml.etree.ElementPath ['/usr/local/lib/python3.3/xml/etree']
Looking for _elementtree None
Looking for copy None
Looking for org None
Looking for pyexpat None
Looking for ElementC14N None
>>>
```

The placement of the finder on `sys.meta_path` is critical. Remove it from the front of the list to the end of the list and try more imports:

```
>>> del sys.meta_path[0]
>>> sys.meta_path.append(Finder())
>>> import urllib.request
>>> import datetime
```

Now you don't see any output because the imports are being handled by other entries in `sys.meta_path`. In this case, you would only see it trigger when nonexistent modules are imported:

```
>>> import fib
Looking for fib None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import xml.superfast
Looking for xml.superfast ['/usr/local/lib/python3.3/xml']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'xml.superfast'
>>>
```

The fact that you can install a finder to catch unknown modules is the key to the `UrlMetaFinder` class in this recipe. An instance of `UrlMetaFinder` is added to the end of `sys.meta_path`, where it serves as a kind of importer of last resort. If the requested module name can't be located by any of the other import mechanisms, it gets handled by this finder. Some care needs to be taken when handling packages. Specifically, the value presented in the path argument needs to be checked to see if it starts with the URL registered in the finder. If not, the submodule must belong to some other finder and should be ignored.

Additional handling of packages is found in the `UrlPackageLoader` class. This class, rather than importing the package name, tries to load the underlying `__init__.py` file. It also sets the module `__path__` attribute. This last part is critical, as the value set will be passed to subsequent `find_module()` calls when loading package submodules. The path-based import hook is an extension of these ideas, but based on a somewhat different mechanism. As you know, `sys.path` is a list of directories where Python looks for modules. For example:

```
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
 '/usr/local/lib/python33.zip',
 '/usr/local/lib/python3.3',
 '/usr/local/lib/python3.3/plat-darwin',
 '/usr/local/lib/python3.3/lib-dynload',
 '/usr/local/lib/...3.3/site-packages']
>>>
```

Each entry in `sys.path` is additionally attached to a finder object. You can view these finders by looking at `sys.path_importer_cache`:

```
>>> pprint(sys.path_importer_cache)
{'.': FileFinder('.'),
 '/usr/local/lib/python3.3': FileFinder('/usr/local/lib/python3.3'),
 '/usr/local/lib/python3.3/': FileFinder('/usr/local/lib/python3.3/'),
 '/usr/local/lib/python3.3/collections': FileFinder('...python3.3/collections'),
 '/usr/local/lib/python3.3/encodings': FileFinder('...python3.3/encodings'),
 '/usr/local/lib/python3.3/lib-dynload': FileFinder('...python3.3/lib-dynload'),
 '/usr/local/lib/python3.3/plat-darwin': FileFinder('...python3.3/plat-darwin'),
 '/usr/local/lib/python3.3/site-packages': FileFinder('...python3.3/site-packages'),
 '/usr/local/lib/python33.zip': None}
>>>
```

`sys.path_importer_cache` tends to be much larger than `sys.path` because it records finders for all known directories where code is being loaded. This includes subdirectories of packages which usually aren't included on `sys.path`.

To execute `import fib`, the directories on `sys.path` are checked in order. For each directory, the name `fib` is presented to the associated finder found in `sys.path_importer_cache`. This is also something that you can investigate by making your own finder and putting an entry in the cache. Try this experiment:

```

>>> class Finder:
...     def find_loader(self, name):
...         print('Looking for', name)
...         return (None, [])
...
>>> import sys
>>> # Add a "debug" entry to the importer cache
>>> sys.path_importer_cache['debug'] = Finder()
>>> # Add a "debug" directory to sys.path
>>> sys.path.insert(0, 'debug')
>>> import threading
Looking for threading
Looking for time
Looking for traceback
Looking for linecache
Looking for tokenize
Looking for token
>>>

```

Here, you've installed a new cache entry for the name debug and installed the name debug as the first entry on sys.path. On all subsequent imports, you see your finder being triggered. However, since it returns (None, []), processing simply continues to the next entry.

The population of sys.path_importer_cache is controlled by a list of functions stored in sys.path_hooks. Try this experiment, which clears the cache and adds a new path checking function to sys.path_hooks:

```

>>> sys.path_importer_cache.clear()
>>> def check_path(path):
...     print('Checking', path)
...     raise ImportError()
...
>>> sys.path_hooks.insert(0, check_path)
>>> import fib
Checked debug
Checking .
Checking /usr/local/lib/python3.3.zip
Checking /usr/local/lib/python3.3
Checking /usr/local/lib/python3.3/plat-darwin
Checking /usr/local/lib/python3.3/lib-dynload
Checking /Users/beazley/.local/lib/python3.3/site-packages
Checking /usr/local/lib/python3.3/site-packages
Looking for fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>>

```

As you can see, the `check_path()` function is being invoked for every entry on `sys.path`. However, since an `ImportError` exception is raised, nothing else happens (checking just moves to the next function on `sys.path_hooks`).

Using this knowledge of how `sys.path` is processed, you can install a custom path checking function that looks for filename patterns, such as URLs. For instance:

```
>>> def check_url(path):
...     if path.startswith('http://'):
...         return Finder()
...     else:
...         raise ImportError()
...
>>> sys.path.append('http://localhost:15000')
>>> sys.path_hooks[0] = check_url
>>> import fib
Looking for fib # Finder output!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Notice installation of Finder in sys.path_importer_cache
>>> sys.path_importer_cache['http://localhost:15000']
<__main__.Finder object at 0x10064c850>
>>>
```

This is the key mechanism at work in the last part of this recipe. Essentially, a custom path checking function has been installed that looks for URLs in `sys.path`. When they are encountered, a new `UrlPathFinder` instance is created and installed into `sys.path_importer_cache`. From that point forward, all import statements that pass through that part of `sys.path` will try to use your custom finder.

Package handling with a path-based importer is somewhat tricky, and relates to the return value of the `find_loader()` method. For simple modules, `find_loader()` returns a tuple (loader, None) where loader is an instance of a loader that will import the module.

For a normal package, `find_loader()` returns a tuple (loader, path) where loader is the loader instance that will import the package (and execute `__init__.py`) and path is a list of the directories that will make up the initial setting of the package's `__path__` attribute. For example, if the base URL was <http://localhost:15000> and a user executed `import grok`, the path returned by `find_loader()` would be [`'http://localhost:15000/grok'`].

The `find_loader()` must additionally account for the possibility of a namespace package. A namespace package is a package where a valid package directory name exists, but no `__init__.py` file can be found. For this case, `find_loader()` must return a tuple (None, path) where path is a list of directories that would have made up the package's `__path__` attribute

had it defined an `__init__.py` file. For this case, the import mechanism moves on to check further directories on `sys.path`. If more namespace packages are found, all of the resulting paths are joined together to make a final namespace package. See Recipe 10.5 for more information on namespace packages.

There is a recursive element to package handling that is not immediately obvious in the solution, but also at work. All packages contain an internal path setting, which can be found in `__path__` attribute. For example:

```
>>> import xml.etree.ElementTree
>>> xml.__path__
['/usr/local/lib/python3.3/xml']
>>> xml.etree.__path__
['/usr/local/lib/python3.3/xml/etree']
>>>
```

As mentioned, the setting of `__path__` is controlled by the return value of the `finder.find_loader()` method. However, the subsequent processing of `__path__` is also handled by the functions in `sys.path_hooks`. Thus, when package subcomponents are loaded, the entries in `__path__` are checked by the `handle_url()` function. This causes new instances of `UrlPathFinder` to be created and added to `sys.path_importer_cache`.

One remaining tricky part of the implementation concerns the behavior of the `handle_url()` function and its interaction with the `_get_links()` function used internally. If your implementation of a finder involves the use of other modules (e.g., `urllib.request`), there is a possibility that those modules will attempt to make further imports in the middle of the finder's operation. This can actually cause `handle_url()` and other parts of the finder to get executed in a kind of recursive loop. To account for this possibility, the implementation maintains a cache of created finders (one per URL). This avoids the problem of creating duplicate finders. In addition, the following fragment of code ensures that the finder doesn't respond to any import requests while it's in the process of getting the initial set of links:

```
# Check link cache
if self._links is None:
    self._links = [] # See discussion
    self._links = _get_links(self._baseurl)
```

You may not need this checking in other implementations, but for this example involving URLs, it was required.

Finally, the `invalidate_caches()` method of both finders is a utility method that is supposed to clear internal caches should the source code change. This method is triggered when a user invokes `importlib.invalidate_caches()`. You might use it if you want the URL importers to reread the list of links, possibly for the purpose of being able to access newly added files.

In comparing the two approaches (modifying `sys.meta_path` or using a path hook), it helps to take a high-level view. Importers installed using `sys.meta_path` are free to handle modules in any manner that they wish. For instance, they could load modules out of a database or import them in a manner that is radically different than normal module/package handling. This freedom also means that such importers need to do more bookkeeping and internal management. This explains, for instance, why the implementation of `UrlMetaFinder` needs to do its own caching of links, loaders, and other details. On the other hand, path-based hooks are more narrowly tied to the processing of `sys.path`. Because of the connection to `sys.path`, modules loaded with such extensions will tend to have the same features as normal modules and packages that programmers are used to.

Assuming that your head hasn't completely exploded at this point, a key to understanding and experimenting with this recipe may be the added logging calls. You can enable logging and try experiments such as this:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> import urlimport
>>> urlimport.install_path_hook()
DEBUG:urlimport:Installing handle_url
>>> import fib
DEBUG:urlimport:Handle path? /usr/local/lib/python33.zip. [No]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
DEBUG:urlimport:Handle path? http://localhost:15000. [Yes]
DEBUG:urlimport:Getting links from http://localhost:15000
DEBUG:urlimport:links: {'spam.py', 'fib.py', 'grok'}
DEBUG:urlimport:find_loader: 'fib'
DEBUG:urlimport:find_loader: module 'fib' found
DEBUG:urlimport:loader: reading 'http://localhost:15000/fib.py'
DEBUG:urlimport:loader: 'http://localhost:15000/fib.py' loaded
I'm fib
>>>
```

Last, but not least, spending some time sleeping with [PEP 302](#) and the documentation for `importlib` under your pillow may be advisable.

10.12 导入模块的同时修改模块

问题

You want to patch or apply decorators to functions in an existing module. However, you only want to do it if the module actually gets imported and used elsewhere.

解决方案

The essential problem here is that you would like to carry out actions in response to a module being loaded. Perhaps you want to trigger some kind of callback function that would notify you when a module was loaded.

This problem can be solved using the same import hook machinery discussed in Recipe 10.11. Here is a possible solution:

```

# postimport.py
import importlib
import sys
from collections import defaultdict

_post_import_hooks = defaultdict(list)

class PostImportFinder:
    def __init__(self):
        self._skip = set()

    def find_module(self, fullname, path=None):
        if fullname in self._skip:
            return None
        self._skip.add(fullname)
        return PostImportLoader(self)

class PostImportLoader:
    def __init__(self, finder):
        self._finder = finder

    def load_module(self, fullname):
        importlib.import_module(fullname)
        module = sys.modules[fullname]
        for func in _post_import_hooks[fullname]:
            func(module)
        self._finder._skip.remove(fullname)
        return module

def when_imported(fullname):
    def decorate(func):
        if fullname in sys.modules:
            func(sys.modules[fullname])
        else:
            _post_import_hooks[fullname].append(func)
        return func
    return decorate

sys.meta_path.insert(0, PostImportFinder())

```

To use this code, you use the `when_imported()` decorator. For example:

```

>>> from postimport import when_imported
>>> @when_imported('threading')
... def warn_threads(mod):
...     print('Threads? Are you crazy?')
...
>>>
>>> import threading
Threads? Are you crazy?
>>>

```

As a more practical example, maybe you want to apply decorators to existing definitions, such as shown here:

```
from functools import wraps
from postimport import when_imported

def logged(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__, args, kwargs)
        return func(*args, **kwargs)
    return wrapper

# Example
@when_imported('math')
def add_logging(mod):
    mod.cos = logged(mod.cos)
    mod.sin = logged(mod.sin)
```

讨论

This recipe relies on the import hooks that were discussed in Recipe 10.11, with a slight twist.

First, the role of the `@when_imported` decorator is to register handler functions that get triggered on import. The decorator checks `sys.modules` to see if a module was already loaded. If so, the handler is invoked immediately. Otherwise, the handler is added to a list in the `_post_import_hooks` dictionary. The purpose of `_post_import_hooks` is simply to collect all handler objects that have been registered for each module. In principle, more than one handler could be registered for a given module.

To trigger the pending actions in `_post_import_hooks` after module import, the `PostImportFinder` class is installed as the first item in `sys.meta_path`. If you recall from Recipe 10.11, `sys.meta_path` contains a list of finder objects that are consulted in order to locate modules. By installing `PostImportFinder` as the first item, it captures all module imports.

In this recipe, however, the role of `PostImportFinder` is not to load modules, but to trigger actions upon the completion of an import. To do this, the actual import is delegated to the other finders on `sys.meta_path`. Rather than trying to do this directly, the function `imp.import_module()` is called recursively in the `PostImportLoader` class. To avoid getting stuck in an infinite loop, `PostImportFinder` keeps a set of all the modules that are currently

in the process of being loaded. If a module name is part of this set, it is simply ignored by PostImportFinder. This is what causes the import request to pass to the other finders on `sys.meta_path`.

After a module has been loaded with `imp.import_module()`, all handlers currently registered in `_post_import_hooks` are called with the newly loaded module as an argument.

From this point forward, the handlers are free to do what they want with the module. A major feature of the approach shown in this recipe is that the patching of a module occurs in a seamless fashion, regardless of where or how a module of interest is actually loaded. You simply write a handler function that's decorated with `@when_imported()` and it all just magically works from that point forward.

One caution about this recipe is that it does not work for modules that have been explicitly reloaded using `imp.reload()`. That is, if you reload a previously loaded module, the post import handler function doesn't get triggered again (all the more reason to not use `reload()` in production code). On the other hand, if you delete the module from `sys.modules` and redo the import, you'll see the handler trigger again.

More information about post-import hooks can be found in PEP 369 . As of this writing, the PEP has been withdrawn by the author due to it being out of date with the current implementation of the `importlib` module. However, it is easy enough to implement your own solution using this recipe.

10.13 安装私有的包

问题

You want to install a third-party package, but you don't have permission to install packages into the system Python. Alternatively, perhaps you just want to install a package for your own use, not all users on the system.

解决方案

Python has a per-user installation directory that's typically located in a directory such as `~/.local/lib/python3.3/site-packages`. To force packages to install in this directory, give the `-user` option to the installation command. For example:

```
python3 setup.py install --user
```

or

```
pip install --user packagename
```

The user site-packages directory normally appears before the system site-packages directory on `sys.path`. Thus, packages you install using this technique take priority over the packages already installed on the system (although this is not always the case depending on the behavior of third-party package managers, such as distribute or pip).

讨论

Normally, packages get installed into the system-wide site-packages directory, which is found in a location such as `/usr/local/lib/python3.3/site-packages`. However, doing so typically requires administrator permissions and use of the `sudo` command. Even if you have permission to execute such a command, using `sudo` to install a new, possibly unproven, package might give you some pause.

Installing packages into the per-user directory is often an effective workaround that allows you to create a custom installation.

As an alternative, you can also create a virtual environment, which is discussed in the next recipe.

10.14 创建新的Python环境

问题

You want to create a new Python environment in which you can install modules and packages. However, you want to do this without installing a new copy of Python or making changes that might affect the system Python installation.

解决方案

You can make a new “virtual” environment using the `pyenv` command. This command is installed in the same directory as the Python interpreter or possibly in the `Scripts` directory on Windows. Here is an example:

```
bash % pyenv Spam
bash %
```

The name supplied to pyenv is the name of a directory that will be created. Upon creation, the Spam directory will look something like this:

```
bash % cd Spam
bash % ls
bin include lib pyenv.cfg
bash %
```

In the bin directory, you'll find a Python interpreter that you can use. For example:

```
bash % Spam/bin/python3
Python 3.3.0 (default, Oct 6 2012, 15:45:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
 '/usr/local/lib/python3.3.zip',
 '/usr/local/lib/python3.3',
 '/usr/local/lib/python3.3/plat-darwin',
 '/usr/local/lib/python3.3/lib-dynload',
 '/Users/beazley/Spam/lib/python3.3/site-packages']
>>>
```

A key feature of this interpreter is that its site-packages directory has been set to the newly created environment. Should you decide to install third-party packages, they will be installed here, not in the normal system site-packages directory.

讨论

The creation of a virtual environment mostly pertains to the installation and management of third-party packages. As you can see in the example, the sys.path variable contains directories from the normal system Python, but the site-packages directory has been relocated to a new directory.

With a new virtual environment, the next step is often to install a package manager, such as `distribute` or `pip`. When installing such tools and subsequent packages, you just need to make sure you use the interpreter that's part of the virtual environment. This should install the packages into the newly created `site-packages` directory.

Although a virtual environment might look like a copy of the Python installation, it really only consists of a few files and symbolic links. All of the standard library files and interpreter executables come from the original Python installation. Thus, creating such environments is easy, and takes almost no machine resources.

By default, virtual environments are completely clean and contain no third-party addons. If you would like to include already installed packages as part of a virtual environment, create the environment using the `--system-site-packages` option. For example:

```
bash % pyenv --system-site-packages Spam
bash %
```

More information about `pyenv` and virtual environments can be found in [PEP 405](#).

10.15 分发包

问题

You've written a useful library, and you want to be able to give it away to others.

解决方案

If you're going to start giving code away, the first thing to do is to give it a unique name and clean up its directory structure. For example, a typical library package might look something like this:


```
projectname/  
  README.txt  
  Doc/  
    documentation.txt  
  projectname/  
    __init__.py  
    foo.py  
    bar.py  
    utils/  
      __init__.py  
      spam.py  
      grok.py  
  examples/  
    helloworld.py  
  ...
```

To make the package something that you can distribute, first write a `setup.py` file that looks like this:

```
# setup.py  
from distutils.core import setup  
  
setup(name='projectname',  
      version='1.0',  
      author='Your Name',  
      author_email='you@youraddress.com',  
      url='http://www.you.com/projectname',  
      packages=['projectname', 'projectname.utils'],  
    )
```

Next, make a file `MANIFEST.in` that lists various nonsource files that you want to include in your package:

```
# MANIFEST.in  
include *.txt  
recursive-include examples *  
recursive-include Doc *
```

Make sure the `setup.py` and `MANIFEST.in` files appear in the top-level directory of your package. Once you have done this, you should be able to make a source distribution by typing a command such as this:

```
% bash python3 setup.py sdist
```

This will create a file such as `projectname-1.0.zip` or `projectname-1.0.tar.gz`, depending on the platform. If it all works, this file is suitable for giving to others or uploading to the [Python Package Index](#).

讨论

For pure Python code, writing a plain `setup.py` file is usually straightforward. One potential gotcha is that you have to manually list every subdirectory that makes up the packages source code. A common mistake is to only list the top-level directory of a package and to forget to include package subcomponents. This is why the specification for packages in `setup.py` includes the `list_packages=['projectname', 'project name.utils']`.

As most Python programmers know, there are many third-party packaging options, including `setuptools`, `distribute`, and so forth. Some of these are replacements for the `distutils` library found in the standard library. Be aware that if you rely on these packages, users may not be able to install your software unless they also install the required package manager first. Because of this, you can almost never go wrong by keeping things as simple as possible. At a bare minimum, make sure your code can be installed using a standard Python 3 installation. Additional features can be supported as an option if additional packages are available.

Packaging and distribution of code involving C extensions can get considerably more complicated. Chapter 15 on C extensions has a few details on this. In particular, see Recipe 15.2.

第十一章：网络与Web编程

本章是关于在网络应用和分布式应用中使用的各种主题。主题划分为使用Python编写客户端程序来访问已有的服务，以及使用Python实现网络服务端程序。也给出了一些常见的技术，用于编写涉及协同或通信的代码。

Contents:

11.1 作为客户端与HTTP服务交互

问题

你需要通过HTTP协议以客户端的方式访问多种服务。例如，下载数据或者与基于REST的API进行交互。

解决方案

对于简单的事情来说，通常使用 `urllib.request` 模块就够了。例如，发送一个简单的 HTTP GET 请求到远程的服务上，可以这样做：

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/get'

# Dictionary of query parameters (if any)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)

# Make a GET request and read the response
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

如果你需要使用 POST 方法在请求主体中发送查询参数，可以将参数编码后作为可选参数提供给 `urlopen()` 函数，就像这样：

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)

# Make a POST request and read the response
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

如果你需要在发出的请求中提供一些自定义的 HTTP 头，例如修改 `user-agent` 字段，可以创建一个包含字段值的字典，并创建一个 Request 实例然后将其传给 `urlopen()`，如下：

```

from urllib import request, parse
...

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

req = request.Request(url, querystring.encode('ascii'), headers=headers)

# Make a request and read the response
u = request.urlopen(req)
resp = u.read()

```

如果需要交互的服务比上面的例子都要复杂，也许应该去看看 `requests` 库 (<https://pypi.python.org/pypi/requests>)。例如，下面这个示例采用 `requests` 库重新实现了上面的操作：

```

import requests

# Base URL being accessed
url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

resp = requests.post(url, data=parms, headers=headers)

# Decoded text returned by the request
text = resp.text

```

关于 `requests` 库，一个值得一提的特性就是它能以多种方式从请求中返回响应结果的内容。从上面的代码来看，`resp.text` 带给我们的是以 `Unicode` 解码的响应文本。但是，如果去访问 `resp.content`，就会得到原始的二进制数据。另一方面，如果访问 `resp.json`，那么就会得到 `JSON` 格式的响应内容。

下面这个示例利用 `requests` 库发起一个 `HEAD` 请求，并从响应中提取出一些 `HTTP` 头数据的字段：

```
import requests
```

```
resp = requests.head('http://www.python.org/index.html')
```

```
status = resp.status_code
```

```
last_modified = resp.headers['last-modified']
```

```
content_type = resp.headers['content-type']
```

```
content_length = resp.headers['content-length']
```

Here is a requests example that executes a login into the Python Package index using basic authentication:

```
import requests
```

```
resp = requests.get('http://pypi.python.org/pypi?:action=login',  
                    auth=('user', 'password'))
```

Here is an example of using requests to pass HTTP cookies from one request to the next:

```
import requests
```

```
# First request
```

```
resp1 = requests.get(url)
```

```
...
```

```
# Second requests with cookies received on first requests
```

```
resp2 = requests.get(url, cookies=resp1.cookies)
```

Last, but not least, here is an example of using requests to upload content:

```
import requests
```

```
url = 'http://httpbin.org/post'
```

```
files = { 'file': ('data.csv', open('data.csv', 'rb')) }
```

```
r = requests.post(url, files=files)
```

讨论

对于真的很简单HTTP客户端代码，用内置的 `urllib` 模块通常就足够了。但是，如果你要做的不仅仅只是简单的GET或POST请求，那就真的不能再依赖它的功能了。这时候就是第三方模块比如 `requests` 大显身手的时候了。

例如，如果你决定坚持使用标准的程序库而不考虑像 `requests` 这样的第三方库，那么也许就不得不使用底层的 `http.client` 模块来实现自己的代码。比方说，下面的代码展示了如何执行一个HEAD请求：

```

from http.client import HTTPConnection
from urllib import parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print('Status', resp.status)
for name, value in resp.getheaders():
    print(name, value)

```

同样地，如果必须编写涉及代理、认证、cookies以及其他一些细节方面的代码，那么使用 `urllib` 就显得特别别扭和啰嗦。比方说，下面这个示例实现在Python包索引上的认证：

```

import urllib.request

auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi', 'http://pypi.python.org', 'username', 'password')
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?action=login')
u = opener.open(r)
resp = u.read()

# From here. You can access more pages using opener
...

```

坦白说，所有的这些操作在 `requests` 库中都变得简单的多。

在开发过程中测试HTTP客户端代码常常是很令人沮丧的，因为所有棘手的细节问题都需要考虑（例如cookies、认证、HTTP头、编码方式等）。要完成这些任务，考虑使用 `httpbin` 服务（<http://httpbin.org>）。这个站点会接收发出的请求，然后以JSON的形式将相应信息回传回来。下面是一个交互式的例子：

```

>>> import requests
>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...                 headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type': '',
'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
'keep-alive', 'Host': 'httpbin.org', 'Accept': '*//*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>

```

在要同一个真正的站点进行交互前，先在 httpbin.org 这样的网站上做实验常常是可取的办法。尤其是当我们面对3次登录失败就会关闭账户这样的风险时尤为有用（不要尝试自己编写HTTP认证客户端来登录你的银行账户）。

尽管本节没有涉及，`request` 库还对许多高级的HTTP客户端协议提供了支持，比如OAuth。`requests` 模块的文档（<http://docs.python-requests.org>）质量很高（坦白说比在这短短的一节的篇幅中所提供的任何信息都好），可以参考文档以获得更多地信息。

11.2 创建TCP服务器

问题

你想实现一个服务器，通过TCP协议和客户端通信。

解决方案

创建一个TCP服务器的一个简单方法是使用 `socketserver` 库。例如，下面是一个简单的应答服务器：

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        while True:

            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

在这段代码中，你定义了一个特殊的处理类，实现了一个 `handle()` 方法，用来为客户端连接服务。`request` 属性是客户端socket，`client_address` 有客户端地址。为了测试这个服务器，运行它并打开另外一个Python进程连接这个服务器：

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
>>> s.recv(8192)
b'Hello'
>>>
```

很多时候，可以很容易的定义一个不同的处理器。下面是一个使用 `StreamRequestHandler` 基类将一个类文件接口放置在底层socket上的例子：

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # self.rfile is a file-like object for reading
        for line in self.rfile:
            # self.wfile is a file-like object for writing
            self.wfile.write(line)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

讨论

`socketserver` 可以让我们很容易的创建简单的TCP服务器。但是，你需要注意的是，默认情况下这种服务器是单线程的，一次只能为一个客户端连接服务。如果你想处理多个客户端，可以初始化一个 `ForkingTCPServer` 或者是 `ThreadingTCPServer` 对象。例如：

```
from socketserver import ThreadingTCPServer

if __name__ == '__main__':
    serv = ThreadingTCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

使用fork或线程服务器有个潜在问题就是它们会为每个客户端连接创建一个新的进程或线程。由于客户端连接数是没有限制的，因此一个恶意的黑客可以同时发送大量的连接让你的服务器崩溃。

如果你担心这个问题，你可以创建一个预先分配大小的工作线程池或进程池。你先创建一个普通的非线程服务器，然后在一个线程池中使用 `serve_forever()` 方法来启动它们。

```
if __name__ == '__main__':
    from threading import Thread
    NWORKERS = 16
    serv = TCPServer('', 20000), EchoHandler)
    for n in range(NWORKERS):
        t = Thread(target=serv.serve_forever)
        t.daemon = True
        t.start()
    serv.serve_forever()
```

一般来讲，一个 `TCPServer` 在实例化的时候会绑定并激活相应的 `socket`。不过，有时候你想通过设置某些选项去调整底下的 `socket`，可以设置参数 `bind_and_activate=False`。如下：

```
if __name__ == '__main__':
    serv = TCPServer('', 20000), EchoHandler, bind_and_activate=False)
    # Set up various socket options
    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    # Bind and activate
    serv.server_bind()
    serv.server_activate()
    serv.serve_forever()
```

上面的 `socket` 选项是一个非常普遍的配置项，它允许服务器重新绑定一个之前使用过的端口号。由于要被经常使用到，它被放置到类变量中，可以直接在 `TCPServer` 上面设置。在实例化服务器的时候去设置它的值，如下所示：

```
if __name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer('', 20000), EchoHandler)
    serv.serve_forever()
```

在上面示例中，我们演示了两种不同的处理器基类（`BaseRequestHandler` 和 `StreamRequestHandler`）。`StreamRequestHandler` 更加灵活点，能通过设置其他的类变量来支持一些新的特性。比如：

```

import socket

class EchoHandler(StreamRequestHandler):
    # Optional settings (defaults shown)
    timeout = 5 # Timeout on all socket operations
    rbufsize = -1 # Read buffer size
    wbufsize = 0 # Write buffer size
    disable_nagle_algorithm = False # Sets TCP_NODELAY socket option
    def handle(self):
        print('Got connection from', self.client_address)
        try:
            for line in self.rfile:
                # self.wfile is a file-like object for writing
                self.wfile.write(line)
        except socket.timeout:
            print('Timed out!')

```

最后，还需要注意的是巨大部分Python的高层网络模块（比如HTTP、XML-RPC等）都是建立在 `socketserver` 功能之上。也就是说，直接使用 `socket` 库来实现服务器也并不是很难。下面是一个使用 `socket` 直接编程实现的一个服务器简单例子：

```

from socket import socket, AF_INET, SOCK_STREAM

def echo_handler(address, client_sock):
    print('Got connection from {}'.format(address))
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)
    client_sock.close()

def echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if __name__ == '__main__':
    echo_server('', 20000)

```

11.3 创建UDP服务器

问题

你想实现一个基于UDP协议的服务器来与客户端通信。

解决方案

跟TCP一样，UDP服务器也可以通过使用 `socketserver` 库很容易的被创建。例如，下面是一个简单的时间服务器：

```
from socketserver import BaseRequestHandler, UDPServer
import time

class TimeHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # Get message and client socket
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    serv = UDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

跟之前一样，你先定义一个实现 `handle()` 特殊方法的类，为客户端连接服务。这个类的 `request` 属性是一个包含了数据报和底层socket对象的元组。 `client_address` 包含了客户端地址。

我们来测试下这个服务器，首先运行它，然后打开另外一个Python进程向服务器发送消息：

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

讨论

一个典型的UDP服务器接收到达的数据报(消息)和客户端地址。如果服务器需要做应答，它要给客户端回发一个数据报。对于数据报的传送，你应该使用socket的 `sendto()` 和 `recvfrom()` 方法。尽管传统的 `send()` 和 `recv()` 也可以达到同样的效果，但是前面的

两个方法对于UDP连接而言更普遍。

由于没有底层的连接，UDP服务器相对于TCP服务器来讲实现起来更加简单。不过，UDP天生是不可靠的（因为通信没有建立连接，消息可能丢失）。因此需要由你自己来决定该怎样处理丢失消息的情况。这个已经不在本书讨论范围内了，不过通常来说，如果可靠性对于你程序很重要，你需要借助于序列号、重试、超时以及一些其他方法来保证。UDP通常被用在那些对于可靠传输要求不是很高的场合。例如，在实时应用如多媒体流以及游戏领域，无需返回恢复丢失的数据包（程序只需简单的忽略它并继续向前运行）。

`UDPServer` 类是单线程的，也就是说一次只能为一个客户端连接服务。实际使用中，这个无论是对于UDP还是TCP都不是什么大问题。如果你想要并发操作，可以实例化一个 `ForkingUDPServer` 或 `ThreadingUDPServer` 对象：

```
from socketserver import ThreadingUDPServer

if __name__ == '__main__':
    serv = ThreadingUDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

直接使用 `socket` 来是想一个UDP服务器也不难，下面是一个例子：

```
from socket import socket, AF_INET, SOCK_DGRAM
import time

def time_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), addr)

if __name__ == '__main__':
    time_server(('', 20000))
```

11.4 通过CIDR地址生成对应的IP地址集

问题

你有一个CIDR网络地址比如“123.45.67.89/27”，你想将其转换成它所代表的所有IP（比如，“123.45.67.64”，“123.45.67.65”，...，“123.45.67.95”）

解决方案

可以使用 `ipaddress` 模块很容易的实现这样的计算。例如：

```
>>> import ipaddress
>>> net = ipaddress.ip_network('123.45.67.64/27')
>>> net
IPv4Network('123.45.67.64/27')
>>> for a in net:
...     print(a)
...
123.45.67.64
123.45.67.65
123.45.67.66
123.45.67.67
123.45.67.68
...
123.45.67.95
>>>

>>> net6 = ipaddress.ip_network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> net6
IPv6Network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> for a in net6:
...     print(a)
...
12:3456:78:90ab:cd:ef01:23:30
12:3456:78:90ab:cd:ef01:23:31
12:3456:78:90ab:cd:ef01:23:32
12:3456:78:90ab:cd:ef01:23:33
12:3456:78:90ab:cd:ef01:23:34
12:3456:78:90ab:cd:ef01:23:35
12:3456:78:90ab:cd:ef01:23:36
12:3456:78:90ab:cd:ef01:23:37
>>>
```

`Network` 也允许像数组一样的索引起值，例如：

```
>>> net.num_addresses
32
>>> net[0]

IPv4Address('123.45.67.64')
>>> net[1]
IPv4Address('123.45.67.65')
>>> net[-1]
IPv4Address('123.45.67.95')
>>> net[-2]
IPv4Address('123.45.67.94')
>>>
```

另外，你还可以执行网络成员检查之类的操作：

```
>>> a = ipaddress.ip_address('123.45.67.69')
>>> a in net
True
>>> b = ipaddress.ip_address('123.45.67.123')
>>> b in net
False
>>>
```

一个IP地址和网络地址能通过一个IP接口来指定，例如：

```
>>> inet = ipaddress.ip_interface('123.45.67.73/27')
>>> inet.network
IPv4Network('123.45.67.64/27')
>>> inet.ip
IPv4Address('123.45.67.73')
>>>
```

讨论

`ipaddress` 模块有很多类可以表示IP地址、网络和接口。当你需要操作网络地址（比如解析、打印、验证等）的时候会很有用。

要注意的是，`ipaddress` 模块跟其他一些和网络相关的模块比如 `socket` 库交集很少。所以，你不能使用 `IPv4Address` 的实例来代替一个地址字符串，你首先得显式的使用 `str()` 转换它。例如：

```
>>> a = ipaddress.ip_address('127.0.0.1')
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect((a, 8080))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'IPv4Address' object to str implicitly
>>> s.connect((str(a), 8080))
>>>
```

更多相关内容，请参考 [An Introduction to the ipaddress Module](#)

11.5 创建一个简单的REST接口

问题

你想使用一个简单的REST接口通过网络远程控制或访问你的应用程序，但是你又不想自己去安装一个完整的web框架。

解决方案

构建一个REST风格的接口最简单的方法是创建一个基于WSGI标准（PEP 3333）的很小的库，下面是一个例子：

```

# resty.py

import cgi

def notfound_404(environ, start_response):
    start_response('404 Not Found', [ ('Content-type', 'text/plain') ])
    return [b'Not Found']

class PathDispatcher:
    def __init__(self):
        self.pathmap = { }

    def __call__(self, environ, start_response):
        path = environ['PATH_INFO']
        params = cgi.FieldStorage(environ['wsgi.input'],
                                  environ=environ)
        method = environ['REQUEST_METHOD'].lower()
        environ['params'] = { key: params.getvalue(key) for key in params }
        handler = self.pathmap.get((method,path), notfound_404)
        return handler(environ, start_response)

    def register(self, method, path, function):
        self.pathmap[method.lower(), path] = function
        return function

```

为了使用这个调度器，你只需要编写不同的处理器，就像下面这样：


```

import time

_hello_resp = '''\
<html>
  <head>
    <title>Hello {name}</title>
  </head>
  <body>
    <h1>Hello {name}!</h1>
  </body>
</html>'''

def hello_world(envIRON, start_response):
    start_response('200 OK', [ ('Content-type', 'text/html') ])
    params = environ['params']
    resp = _hello_resp.format(name=params.get('name'))
    yield resp.encode('utf-8')

_localtime_resp = '''\
<?xml version="1.0"?>
<time>
  <year>{t.tm_year}</year>
  <month>{t.tm_mon}</month>
  <day>{t.tm_mday}</day>
  <hour>{t.tm_hour}</hour>
  <minute>{t.tm_min}</minute>
  <second>{t.tm_sec}</second>
</time>'''

def localtime(envIRON, start_response):
    start_response('200 OK', [ ('Content-type', 'application/xml') ])
    resp = _localtime_resp.format(t=time.localtime())
    yield resp.encode('utf-8')

if __name__ == '__main__':
    from resty import PathDispatcher
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    dispatcher.register('GET', '/hello', hello_world)
    dispatcher.register('GET', '/localtime', localtime)

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()

```

要测试下这个服务器，你可以使用一个浏览器或 `urllib` 和它交互。例如：

```
>>> u = urlopen('http://localhost:8080/hello?name=Guido')
>>> print(u.read().decode('utf-8'))
<html>
  <head>
    <title>Hello Guido</title>
  </head>
  <body>
    <h1>Hello Guido!</h1>
  </body>
</html>

>>> u = urlopen('http://localhost:8080/localtime')
>>> print(u.read().decode('utf-8'))
<?xml version="1.0"?>
<time>
  <year>2012</year>
  <month>11</month>
  <day>24</day>
  <hour>14</hour>
  <minute>49</minute>
  <second>17</second>
</time>
>>>
```

讨论

在编写REST接口时，通常都是服务于普通的HTTP请求。但是跟那些功能完整的网站相比，你通常只需要处理数据。这些数据以各种标准格式编码，比如XML、JSON或CSV。尽管程序看上去很简单，但是以这种方式提供的API对于很多应用程序来讲是非常有用的。

例如，长期运行的程序可能会使用一个REST API来实现监控或诊断。大数据应用程序可以使用REST来构建一个数据查询或提取系统。REST还能用来控制硬件设备比如机器人、传感器、工厂或灯泡。更重要的是，REST API已经被大量客户端编程环境所支持，比如Javascript, Android, iOS等。因此，利用这种接口可以让你开发出更加复杂的应用程序。

为了实现一个简单的REST接口，你只需让你的程序代码满足Python的WSGI标准即可。WSGI被标准库支持，同时也被绝大部分第三方web框架支持。因此，如果你的代码遵循这个标准，在后面的使用过程中就会更加的灵活！

在WSGI中，你可以像下面这样约定的方式以一个可调用对象形式来实现你的程序。

```
import cgi

def wsgi_app(environ, start_response):
    pass
```

`environ` 属性是一个字典，包含了从web服务器如Apache[参考Internet RFC 3875]提供的CGI接口中获取的值。要将这些不同的值提取出来，你可以像这么这样写：

```
def wsgi_app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    # Parse the query parameters
    params = cgi.FieldStorage(environ['wsgi.input'], environ=environ)
```

我们展示了一些常见的值。`environ['REQUEST_METHOD']` 代表请求类型如GET、POST、HEAD等。`environ['PATH_INFO']` 表示被请求资源的路径。调用 `cgi.FieldStorage()` 可以从请求中提取查询参数并将它们放入一个类字典对象中以便后面使用。

`start_response` 参数是一个为了初始化一个请求对象而必须被调用的函数。第一个参数是返回的HTTP状态值，第二个参数是一个(名,值)元组列表，用来构建返回的HTTP头。例如：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
```

为了返回数据，一个WSGI程序必须返回一个字节字符串序列。可以像下面这样使用一个列表来完成：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
    resp = []
    resp.append(b'Hello World\n')
    resp.append(b'Goodbye!\n')
    return resp
```

或者，你还可以使用 `yield`：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
    yield b'Hello World\n'
    yield b'Goodbye!\n'
```

这里要强调的一点是最后返回的必须是字节字符串。如果返回结果包含文本字符串，必须先将其编码成字节。当然，并没有要求你返回的一点是文本，你可以很轻松的编写一个生成图片的程序。

尽管WSGI程序通常被定义成一个函数，不过你也可以使用类实例来实现，只要它实现了合适的 `__call__()` 方法。例如：

```
class WSGIApplication:
    def __init__(self):
        ...
    def __call__(self, environ, start_response)
        ...
```

我们已经在上面使用这种技术创建 `PathDispatcher` 类。这个分发器仅仅只是管理一个字典，将(方法,路径)对映射到处理器函数上面。当一个请求到来时，它的方法和路径被提取出来，然后被分发到对应的处理器上面去。另外，任何查询变量会被解析后放到一个字典中，以 `environ['params']` 形式存储。后面这个步骤太常见，所以建议你在分发器里面完成，这样可以省掉很多重复代码。使用分发器的时候，你只需简单的创建一个实例，然后通过它注册各种WSGI形式的函数。编写这些函数应该超级简单了，只要你遵循 `start_response()` 函数的编写规则，并且最后返回字节字符串即可。

当编写这种函数的时候还需注意的一点就是对于字符串模板的使用。没人愿意写那种到处混合着 `print()` 函数、XML和大量格式化操作的代码。我们上面使用了三引号包含的预先定义好的字符串模板。这种方式的可以让我们很容易的在以后修改输出格式(只需要修改模板本身，而不用动任何使用它的地方)。

最后，使用WSGI还有一个很重要的部分就是没有什么地方是针对特定web服务器的。因为标准对于服务器和框架是中立的，你可以将你的程序放入任何类型服务器中。我们使用下面的代码测试测试本节代码：

```

if __name__ == '__main__':
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    pass

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()

```

上面代码创建了一个简单的服务器，然后你就可以来测试下你的实现是否能正常工作。最后，当你准备进一步扩展你的程序的时候，你可以修改这个代码，让它可以为特定服务器工作。

WSGI本身是一个很小的标准。因此它并没有提供一些高级的特性比如认证、cookies、重定向等。这些你自己实现起来也不难。不过如果你想要更多的支持，可以考虑第三方库，比如 [WebOb](#) 或者 [Paste](#) 。

11.6 通过XML-RPC实现简单的远程调用

问题

You want an easy way to execute functions or methods in Python programs running on remote machines.

解决方案

Perhaps the easiest way to implement a simple remote procedure call mechanism is to use XML-RPC. Here is an example of a simple server that implements a simple key-value store:

```

from xmlrpc.server import SimpleXMLRPCServer

```

```

class KeyValueServer:

```

```

    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, address):

```

```

        self._data = {}
        self._serv = SimpleXMLRPCServer(address, allow_none=True)
        for name
        in self._rpc_methods_:

```

```

            self._serv.register_function(getattr(self, name))

```

```

    def get(self, name):
        return self._data[name]

```

```

def set(self, name, value):
    self._data[name] = value

def delete(self, name):
    del self._data[name]

def exists(self, name):
    return name in self._data

def keys(self):
    return list(self._data)

def serve_forever(self):
    self._serv.serve_forever()

```

Example if `__name__ == '__main__':`

```

kvserv = KeyValueServer(('', 15000)) kvserv.serve_forever()

```

Here is how you would access the server remotely from a client:

```

>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('http://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>

```

讨论

XML-RPC can be an extremely easy way to set up a simple remote procedure call service. All you need to do is create a server instance, register functions with it using the `register_function()` method, and then launch it using the `serve_forever()` method. This recipe packages it up into a class to put all of the code together, but there is no such requirement. For example, you could create a server by trying something like this:

```

from xmlrpc.server import SimpleXMLRPCServer
def add(x,y):
    return x+y

```

```
serv = SimpleXMLRPCServer(('', 15000)) serv.register_function(add) serv.serve_forever()
```

Functions exposed via XML-RPC only work with certain kinds of data such as strings, numbers, lists, and dictionaries. For everything else, some study is required. For instance, if you pass an instance through XML-RPC, only its instance dictionary is handled:

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> p = Point(2, 3)
>>> s.set('foo', p)
>>> s.get('foo')
{'x': 2, 'y': 3}
>>>
```

Similarly, handling of binary data is a bit different than you expect:

```
>>> s.set('foo', b'Hello World')
>>> s.get('foo')
<xmlrpc.client.Binary object at 0x10131d410>
```

```
>>> _.data
b'Hello World'
>>>
```

As a general rule, you probably shouldn't expose an XML-RPC service to the rest of the world as a public API. It often works best on internal networks where you might want to write simple distributed programs involving a few different machines. A downside to XML-RPC is its performance. The SimpleXMLRPCServer implementation is only single threaded, and wouldn't be appropriate for scaling a large application, although it can be made to run multithreaded, as shown in Recipe 11.2. Also, since XML-RPC serializes all data as XML, it's inherently slower than other approaches. However, one benefit of this encoding is that it's understood by a variety of other programming languages. By using it, clients written in languages other than Python will be able to access your service. Despite its limitations, XML-RPC is worth knowing about if you ever have the need to make a quick and dirty remote procedure call system. Oftentimes, the simple solution is good enough.

11.7 在不同的Python解释器之间交互

问题

You are running multiple instances of the Python interpreter, possibly on different machines, and you would like to exchange data between interpreters using messages.

解决方案

It is easy to communicate between interpreters if you use the `multiprocessing.connection` module. Here is a simple example of writing an echo server:

```
from multiprocessing.connection import Listener import traceback
```

```
def echo_client(conn):
    try:
        while True:
            msg = conn.recv() conn.send(msg)

    except EOFError:
        print('Connection closed')

def echo_server(address, authkey):
    serv = Listener(address, authkey=authkey) while True:

        try:
            client = serv.accept()

            echo_client(client)

        except Exception:
            traceback.print_exc()
```

```
echo_server(('', 25000), authkey=b'peekaboo')
```

Here is a simple example of a client connecting to the server and sending various messages:


```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 25000), authkey=b'peekaboo')
>>> c.send('hello')
>>> c.recv()
'hello'
>>> c.send(42)
>>> c.recv()
42
>>> c.send([1, 2, 3, 4, 5])
>>> c.recv()
[1, 2, 3, 4, 5]
>>>
```

Unlike a low-level socket, messages are kept intact (each object sent using `send()` is received in its entirety with `recv()`). In addition, objects are serialized using pickle. So, any object compatible with pickle can be sent or received over the connection.

讨论

There are many packages and libraries related to implementing various forms of message passing, such as ZeroMQ, Celery, and so forth. As an alternative, you might also be inclined to implement a message layer on top of low-level sockets. However, sometimes you just want a simple solution. The `multiprocessing.connection` library is just that—using a few simple primitives, you can easily connect interpreters together and have them exchange messages. If you know that the interpreters are going to be running on the same machine, you can use alternative forms of networking, such as UNIX domain sockets or Windows named pipes. To create a connection using a UNIX domain socket, simply change the address to a filename such as this:

```
s = Listener('/tmp/myconn', authkey=b'peekaboo')
```

To create a connection using a Windows named pipe, use a filename such as this:

```
s = Listener(r'\\pipemyconn', authkey=b'peekaboo')
```

As a general rule, you would not be using multiprocessing to implement public-facing services. The `authkey` parameter to `Client()` and `Listener()` is there to help authenticate the end points of the connection. Connection attempts with a bad key raise an exception. In addition, the module is probably best suited for long-running connections

(not a large number of short connections). For example, two interpreters might establish a connection at startup and keep the connection active for the entire duration of a problem. Don't use multiprocessing if you need more low-level control over aspects of the connection. For example, if you needed to support timeouts, nonblocking I/O, or anything similar, you're probably better off using a different library or implementing such features on top of sockets instead.

11.8 实现远程方法调用

问题

You want to implement simple remote procedure call (RPC) on top of a message passing layer, such as sockets, multiprocessing connections, or ZeroMQ.

解决方案

RPC is easy to implement by encoding function requests, arguments, and return values using pickle, and passing the pickled byte strings between interpreters. Here is an example of a simple RPC handler that could be incorporated into a server:

```
# rpcserver.py
```

```
import pickle class RPCHandler:
```

```

def __init__(self):
    self._functions = {}

def register_function(self, func):
    self._functions[func.__name__] = func

def handle_connection(self, connection):
    try:
        while True:
            # Receive a message func_name, args, kwargs = pickle.loads(connection.recv())
            # Run the RPC and send a response try:

            r = self._functions[func_name](*args,**kwargs)
            connection.send(pickle.dumps(r))

        except Exception as e:
            connection.send(pickle.dumps(e))

    except EOFError:
        pass

```

To use this handler, you need to add it into a messaging server. There are many possible choices, but the multiprocessing library provides a simple option. Here is an example RPC server:

```

from multiprocessing.connection import Listener
from threading import Thread

```

```

def rpc_server(handler, address, authkey):
    sock = Listener(address, authkey=authkey)
    while True:
        client = sock.accept()
        t = Thread(target=handler.handle_connection, args=(client,))
        t.daemon = True
        t.start()

```

```

# Some remote functions
def add(x, y):

```

```

    return x + y

```

```

def sub(x, y):
    return x - y

```

```

# Register with a handler
handler = RPCHandler()
handler.register_function(add)
handler.register_function(sub)

```

```

# Run the server
rpc_server(handler, ('localhost', 17000), authkey=b'peekaboo')

```

To access the server from a remote client, you need to create a corresponding RPC proxy class that forwards requests. For example:

```
import pickle
```

```
class RPCProxy:
```

```
    def __init__(self, connection):
        self._connection = connection
```

```
    def __getattr__(self, name):
```

```
        def do_rpc(*args, **kwargs):
            self._connection.send(pickle.dumps((name, args, kwargs))) result =
            pickle.loads(self._connection.recv()) if isinstance(result, Exception):
```

```
                raise result
```

```
            return result
```

```
        return do_rpc
```

To use the proxy, you wrap it around a connection to the server. For example:

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 17000), authkey=b'peekaboo')
>>> proxy = RPCProxy(c)
>>> proxy.add(2, 3)
```

```
5 >>> proxy.sub(2, 3) -1 >>> proxy.sub([1, 2], 4) Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module> File "rpcserver.py", line 37, in do_rpc
```

```
    raise result
```

```
TypeError: unsupported operand type(s) for -: 'list' and 'int' >>>
```

It should be noted that many messaging layers (such as multiprocessing) already serialize data using pickle. If this is the case, the pickle.dumps() and pickle.loads() calls can be eliminated.

讨论

The general idea of the `RPCHandler` and `RPCProxy` classes is relatively simple. If a client wants to call a remote function, such as `foo(1, 2, z=3)`, the proxy class creates a tuple (`'foo', (1, 2), {'z': 3}`) that contains the function name and arguments. This tuple is pickled and sent over the connection. This is performed in the `do_rpc()` closure that's returned by the `__getattr__()` method of `RPCProxy`. The server receives and unpickles the message, looks up the function name to see if it's registered, and executes it with the given arguments. The result (or exception) is then pickled and sent back. As shown, the example relies on multiprocessing for communication. However, this approach could be made to work with just about any other messaging system. For example, if you want to implement RPC over ZeroMQ, just replace the connection objects with an appropriate ZeroMQ socket object. Given the reliance on pickle, security is a major concern (because a clever hacker can create messages that make arbitrary functions execute during unpickling). In particular, you should never allow RPC from untrusted or unauthenticated clients. In particular, you definitely don't want to allow access from just any machine on the Internet—this should really only be used internally, behind a firewall, and not exposed to the rest of the world. As an alternative to pickle, you might consider the use of JSON, XML, or some other data encoding for serialization. For example, this recipe is fairly easy to adapt to JSON encoding if you simply replace `pickle.loads()` and `pickle.dumps()` with `json.loads()` and `json.dumps()`. For example:

```
# jsonrpcserver.py import json
```

```
class RPCHandler:
```

```
    def __init__(self):
```

```
        self._functions = { }
```

```
    def register_function(self, func):
```

```
        self._functions[func.__name__] = func
```

```
    def handle_connection(self, connection):
```

```
        try:
```

```
            while True:
```

```
                # Receive a message func_name, args, kwargs = json.loads(connection.recv())
```

```
                # Run the RPC and send a response try:
```

```
                    r = self._functions[func_name>(*args,**kwargs)
```

```
                    connection.send(json.dumps(r))
```

```
            except Exception as e:
```

```
                connection.send(json.dumps(str(e)))
```

```
        except EOFError:
```

```
            pass
```

```
# jsonrpcclient.py import json
```

```
class RPCProxy:
```

```

def __init__(self, connection):
    self._connection = connection

def __getattr__(self, name):
    def do_rpc(*args, **kwargs):
        self._connection.send(json.dumps((name, args, kwargs))) result =
        json.loads(self._connection.recv()) return result

    return do_rpc

```

One complicated factor in implementing RPC is how to handle exceptions. At the very least, the server shouldn't crash if an exception is raised by a method. However, the means by which the exception gets reported back to the client requires some study. If you're using pickle, exception instances can often be serialized and reraised in the client. If you're using some other protocol, you might have to think of an alternative approach. At the very least, you would probably want to return the exception string in the response. This is the approach taken in the JSON example. For another example of an RPC implementation, it can be useful to look at the implementation of the SimpleXMLRPCServer and ServerProxy classes used in XML-RPC, as described in Recipe 11.6.

11.9 简单的客户端认证

问题

You want a simple way to authenticate the clients connecting to servers in a distributed system, but don't need the complexity of something like SSL.

解决方案

Simple but effective authentication can be performed by implementing a connection handshake using the hmac module. Here is sample code:

```

import hmac import os

def client_authenticate(connection, secret_key):
    """ Authenticate client to a remote service. connection represents a network connection.
    secret_key is a key known only to both client/server. """ message = connection.recv(32)
    hash = hmac.new(secret_key, message) digest = hash.digest() connection.send(digest)

def server_authenticate(connection, secret_key):
    """ Request client authentication. """ message = os.urandom(32)
    connection.send(message) hash = hmac.new(secret_key, message) digest = hash.digest()
    response = connection.recv(len(digest)) return hmac.compare_digest(digest,response)

```

The general idea is that upon connection, the server presents the client with a message of random bytes (returned by `os.urandom()`, in this case). The client and server both compute a cryptographic hash of the random data using `hmac` and a secret key known only to both ends. The client sends its computed digest back to the server, where it is compared and used to decide whether or not to accept or reject the connection. Comparison of resulting digests should be performed using the `hmac.compare_digest()` function. This function has been written in a way that avoids timing-analysis-based attacks and should be used instead of a normal comparison operator (`==`). To use these functions, you would incorporate them into existing networking or messaging code. For example, with sockets, the server code might look something like this:

```
from socket import socket, AF_INET, SOCK_STREAM
```

```
secret_key = b'peekaboo' def echo_handler(client_sock):
```

```
    if not server_authenticate(client_sock, secret_key):
```

```
        client_sock.close() return
```

```
    while True:
```

```
        msg = client_sock.recv(8192) if not msg:
```

```
            break
```

```
        client_sock.sendall(msg)
```

```
def echo_server(address):
```

```
    s = socket(AF_INET, SOCK_STREAM) s.bind(address) s.listen(5) while True:
```

```
        c,a = s.accept() echo_handler(c)
```

```
echo_server(('', 18000))
```

Within a client, you would do this:

```
from socket import socket, AF_INET, SOCK_STREAM
```

```
secret_key = b'peekaboo'
```

```
s = socket(AF_INET, SOCK_STREAM) s.connect(('localhost', 18000)) client_authenticate(s, secret_key) s.send(b'Hello World') resp = s.recv(1024) ...
```

讨论

A common use of hmac authentication is in internal messaging systems and interprocess communication. For example, if you are writing a system that involves multiple processes communicating across a cluster of machines, you can use this approach to make sure that only allowed processes are allowed to connect to one another. In fact, HMAC-based authentication is used internally by the multiprocessing library when it sets up communication with subprocesses. It's important to stress that authenticating a connection is not the same as encryption. Subsequent communication on an authenticated connection is sent in the clear, and would be visible to anyone inclined to sniff the traffic (although the secret key known to both sides is never transmitted). The authentication algorithm used by hmac is based on cryptographic hashing functions, such as MD5 and SHA-1, and is described in detail in IETF RFC 2104.

11.10 在网络服务中加入SSL

问题

You want to implement a network service involving sockets where servers and clients authenticate themselves and encrypt the transmitted data using SSL.

解决方案

The ssl module provides support for adding SSL to low-level socket connections. In particular, the `ssl.wrap_socket()` function takes an existing socket and wraps an SSL layer around it. For example, here's an example of a simple echo server that presents a server certificate to connecting clients:

```
from socket import socket, AF_INET, SOCK_STREAM import ssl
```

```
KEYFILE = 'server_key.pem' # Private key of the server  
CERTFILE = 'server_cert.pem' #  
Server certificate (given to client)
```

```
def echo_client(s):  
    while True:  
        data = s.recv(8192) if data == b"":  
            break  
  
        s.send(data)  
  
    s.close() print('Connection closed')
```



```

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM) s.bind(address) s.listen(1)

    # Wrap with an SSL layer requiring client certs s_ssl = ssl.wrap_socket(s,
        keyfile=KEYFILE, certfile=CERTFILE, server_side=True)

    # Wait for connections while True:

    try:
        c,a = s_ssl.accept() print('Got connection', c, a) echo_client(c)

    except Exception as e:
        print('{}: {}'.format(e.__class__.__name__, e))

echo_server(('', 20000))

```

Here's an interactive session that shows how to connect to the server as a client. The client requires the server to present its certificate and verifies it:

```

>>> from socket import socket, AF_INET, SOCK_STREAM
>>> import ssl
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s_ssl = ssl.wrap_socket(s,
...                         cert_reqs=ssl.CERT_REQUIRED,
...                         ca_certs = 'server_cert.pem')
>>> s_ssl.connect(('localhost', 20000))
>>> s_ssl.send(b'Hello World?')
12
>>> s_ssl.recv(8192)
b'Hello World?'
>>>

```

The problem with all of this low-level socket hacking is that it doesn't play well with existing network services already implemented in the standard library. For example, most server code (HTTP, XML-RPC, etc.) is actually based on the socketserver library. Client code is also implemented at a higher level. It is possible to add SSL to existing services, but a slightly different approach is needed. First, for servers, SSL can be added through the use of a mixin class like this:

```
import ssl
```

```
class SSLMixin:
```

```
    """ Mixin class that adds support for SSL to existing servers based on the socketserver
    module. """ def __init__(self, *args,
```

```

keyfile=None, certfile=None, ca_certs=None, cert_reqs=ssl.NONE, **kwargs):

self._keyfile = keyfile self._certfile = certfile self._ca_certs = ca_certs self._cert_reqs =
cert_reqs super().__init__(*args, **kwargs)

```

```

def get_request(self):

```

```

    client, addr = super().get_request() client_ssl = ssl.wrap_socket(client,

```

```

        keyfile = self._keyfile, certfile = self._certfile, ca_certs = self._ca_certs, cert_reqs =
        self._cert_reqs, server_side = True)

```

```

    return client_ssl, addr

```

To use this mixin class, you can mix it with other server classes. For example, here's an example of defining an XML-RPC server that operates over SSL:

```

# XML-RPC server with SSL

```

```

from xmlrpc.server import SimpleXMLRPCServer

```

```

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):

```

```

    pass

```

Here's the XML-RPC server from Recipe 11.6 modified only slightly to use SSL:

```

import ssl from xmlrpc.server import SimpleXMLRPCServer from sslmixin import SSLMixin

```

```

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):

```

```

    pass

```

```

class KeyValueServer:

```

```

    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys'] def __init__(self, *args, **kwargs):

```

```

        self._data = {} self._serv = SSLSimpleXMLRPCServer(*args, allow_none=True,
        **kwargs) for name in self._rpc_methods_:

```

```

            self._serv.register_function(getattr(self, name))

```

```

def get(self, name):

```

```

    return self._data[name]

```

```

def set(self, name, value):

```

```

    self._data[name] = value

```

```

def delete(self, name):

```

```

    del self._data[name]

```

```

def exists(self, name):

```

```

    return name in self._data

```

```

def keys(self):
    return list(self._data)

def serve_forever(self):
    self._serv.serve_forever()

if __name__ == '__main__':
    KEYFILE='server_key.pem' # Private key of the server CERTFILE='server_cert.pem' #
    Server certificate
    kvserv = KeyValueServer(('', 15000),

        keyfile=KEYFILE, certfile=CERTFILE),

    kvserv.serve_forever()

```

To use this server, you can connect using the normal `xmlrpc.client` module. Just specify a `https:` in the URL. For example:

```

>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('https://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>

```

One complicated issue with SSL clients is performing extra steps to verify the server certificate or to present a server with client credentials (such as a client certificate). Unfortunately, there seems to be no standardized way to accomplish this, so research is often required. However, here is an example of how to set up a secure XML-RPC connection that verifies the server's certificate:

```

from xmlrpc.client import SafeTransport, ServerProxy
import ssl

```

```

class VerifyCertSafeTransport(SafeTransport):
    def __init__(self, cafile, certfile=None, keyfile=None):
        SafeTransport.__init__(self)
        self._ssl_context =
            ssl.SSLContext(ssl.PROTOCOL_TLSv1)
        self._ssl_context.load_verify_locations(cafile)
        if cert:
            self._ssl_context.load_cert_chain(certfile, keyfile)

```

```
self._ssl_context.verify_mode = ssl.CERT_REQUIRED
```

```
def make_connection(self, host):
```

```
    # Items in the passed dictionary are passed as keyword # arguments to the
    http.client.HTTPSConnection() constructor. # The context argument allows an
    ssl.SSLContext instance to # be passed with information about the SSL configuration
    s = super().make_connection((host, {'context': self._ssl_context}))
```

```
    return s
```

```
# Create the client proxy s = ServerProxy('https://localhost:15000',
```

```
    transport=VerifyCertSafeTransport('server_cert.pem'), allow_none=True)
```

As shown, the server presents a certificate to the client and the client verifies it. This verification can go both directions. If the server wants to verify the client, change the server startup to the following: if `__name__ == '__main__':`

```
KEYFILE='server_key.pem' # Private key of the server CERTFILE='server_cert.pem' #
Server certificate CA_CERTS='client_cert.pem' # Certificates of accepted clients
```

```
kvserv = KeyValueServer(('', 15000),
    keyfile=KEYFILE, certfile=CERTFILE, ca_certs=CA_CERTS,
    cert_reqs=ssl.CERT_REQUIRED,)
```

```
kvserv.serve_forever()
```

To make the XML-RPC client present its certificates, change the `ServerProxy` initialization to this:

```
# Create the client proxy s = ServerProxy('https://localhost:15000',
```

```
    transport=VerifyCertSafeTransport('server_cert.pem',
    'client_cert.pem', 'client_key.pem'),
```

```
    allow_none=True)
```

讨论

Getting this recipe to work will test your system configuration skills and understanding of SSL. Perhaps the biggest challenge is simply getting the initial configuration of keys, certificates, and other matters in order. To clarify what's required, each endpoint of an SSL

connection typically has a private key and a signed certificate file. The certificate file contains the public key and is presented to the remote peer on each connection. For public servers, certificates are normally signed by a certificate authority such as Verisign, Equifax, or similar organization (something that costs money). To verify server certificates, clients maintain a file containing the certificates of trusted certificate authorities. For example, web browsers maintain certificates corresponding to the major certificate authorities and use them to verify the integrity of certificates presented by web servers during HTTPS connections. For the purposes of this recipe, you can create what's known as a self-signed certificate. Here's how you do it:

```
bash % openssl req -new -x509 -days 365 -nodes -out server_cert.pem  
-keyout server_key.pem
```

```
Generating a 1024 bit RSA private key .....++++++ ..++++++
```

writing new private key to 'server_key.pem'

You are about to be asked to enter information that will be incorporated into your certificate request. What you are about to enter is what is called a Distinguished Name or a DN. There are quite a few fields but you can leave some blank For some fields there will be a default value, If you enter '.', the field will be left blank.

```
Country Name (2 letter code) [AU]:US State or Province Name (full name) [Some-  
State]:Illinois Locality Name (eg, city) []:Chicago Organization Name (eg, company)  
[Internet Widgits Pty Ltd]:Dabeaz, LLC Organizational Unit Name (eg, section) []: Common  
Name (eg, YOUR name) []:localhost Email Address []: bash %
```

When creating the certificate, the values for the various fields are often arbitrary. However, the "Common Name" field often contains the DNS hostname of servers. If you're just testing things out on your own machine, use "localhost." Otherwise, use the domain name of the machine that's going to run the server. As a result of this configuration, you will have a server_key.pem file that contains the private key. It looks like this:

certificate on the client machine and use that as a means for verification. During connection, the server will present its certificate, and then you'll use the stored certificate you already have to verify that it's correct. Servers can also elect to verify the identity of clients. To do that, clients need to have their own private key and certificate key. The server would also need to maintain a file of trusted certificate authorities for verifying the client certificates. If you intend to add SSL support to a network service for real, this recipe really only gives a small taste of how to set it up. You will definitely want to consult the documentation for more of the finer points. Be prepared to spend a significant amount of time experimenting with it to get things to work.

11.11 进程间传递Socket文件描述符

问题

You have multiple Python interpreter processes running and you want to pass an open file descriptor from one interpreter to the other. For instance, perhaps there is a server process that is responsible for receiving connections, but the actual servicing of clients is to be handled by a different interpreter.

解决方案

To pass a file descriptor between processes, you first need to connect the processes together. On Unix machines, you might use a Unix domain socket, whereas on Windows, you could use a named pipe. However, rather than deal with such low-level mechanics, it is often easier to use the multiprocessing module to set up such a connection.

Once a connection is established, you can use the `send_handle()` and `recv_handle()` functions in `multiprocessing.reduction` to send file descriptors between processes. The following example illustrates the basics:

```
import multiprocessing from multiprocessing.reduction import recv_handle, send_handle
import socket
```

```
def worker(in_p, out_p):
    out_p.close() while True:
```

```
fd = recv_handle(in_p) print('CHILD: GOT FD', fd) with
socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as s:
```

```
    while True:
```

```
        msg = s.recv(1024) if not msg:
```

```
            break
```

```
        print('CHILD: RECV {!r}'.format(msg)) s.send(msg)
```

```
def server(address, in_p, out_p, worker_pid):
```

```
    in_p.close() s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True) s.bind(address)
```

```
    s.listen(1) while True:
```

```
        client, addr = s.accept() print('SERVER: Got connection from', addr)
```

```
        send_handle(out_p, client.fileno(), worker_pid) client.close()
```

```
if __name__ == '__main__':
```

```
    c1, c2 = multiprocessing.Pipe() worker_p = multiprocessing.Process(target=worker,
    args=(c1,c2)) worker_p.start()
```

```
    server_p = multiprocessing.Process(target=server,
    args=("", 15000), c1, c2, worker_p.pid)
```

```
    server_p.start()
```

```
    c1.close() c2.close()
```

In this example, two processes are spawned and connected by a multiprocessing Pipe object. The server process opens a socket and waits for client connections. The worker process merely waits to receive a file descriptor on the pipe using `recv_handle()`. When the server receives a connection, it sends the resulting socket file descriptor to the worker

using `send_handle()`. The worker takes over the socket and echoes data back to the client until the connection is closed. If you connect to the running server using Telnet or a similar tool, here is an example of what you might see:

```
bash % python3 passfd.py SERVER: Got connection from ('127.0.0.1', 55543) CHILD:
GOT FD 7 CHILD: RECV b'Hellorn' CHILD: RECV b'Worldrn'
```

The most important part of this example is the fact that the client socket accepted in the server is actually serviced by a completely different process. The server merely hands it off, closes it, and waits for the next connection.

讨论

Passing file descriptors between processes is something that many programmers don't even realize is possible. However, it can sometimes be a useful tool in building scalable systems. For example, on a multicore machine, you could have multiple instances of the Python interpreter and use file descriptor passing to more evenly balance the number of clients being handled by each interpreter. The `send_handle()` and `recv_handle()` functions shown in the solution really only work with multiprocessing connections. Instead of using a pipe, you can connect in - terpreters as shown in Recipe 11.7, and it will work as long as you use UNIX domain sockets or Windows pipes. For example, you could implement the server and worker as completely separate programs to be started separately. Here is the implementation of the server:

```
# servermp.py from multiprocessing.connection import Listener from
multiprocessing.reduction import send_handle import socket

def server(work_address, port):
    # Wait for the worker to connect work_serv = Listener(work_address,
    authkey=b'peekaboo') worker = work_serv.accept() worker_pid = worker.recv()

    # Now run a TCP/IP server and send clients to worker s = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM) s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
    True) s.bind(('', port)) s.listen(1) while True:

        client, addr = s.accept() print('SERVER: Got connection from', addr)

        send_handle(worker, client.fileno(), worker_pid) client.close()

if __name__ == '__main__':
    import sys if len(sys.argv) != 3:

        print('Usage: server.py server_address port', file=sys.stderr) raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))
```

To run this server, you would run a command such as `python3 servermp.py /tmp/ servconn 15000`. Here is the corresponding client code:

```
# workermp.py

from multiprocessing.connection import Client from multiprocessing.reduction import
recv_handle import os from socket import socket, AF_INET, SOCK_STREAM

def worker(server_address):
    serv = Client(server_address, authkey=b'peekaboo') serv.send(os.getpid()) while True:
```

```
fd = recv_handle(serv) print('WORKER: GOT FD', fd) with socket(AF_INET,
SOCK_STREAM, fileno=fd) as client:
```

```
    while True:
```

```
        msg = client.recv(1024) if not msg:
```

```
            break
```

```
        print('WORKER: RECV {!r}'.format(msg)) client.send(msg)
```

```
if __name__ == '__main__':
```

```
    import sys if len(sys.argv) != 2:
```

```
        print('Usage: worker.py server_address', file=sys.stderr) raise SystemExit(1)
```

```
    worker(sys.argv[1])
```

To run the worker, you would type `python3 workermp.py /tmp/servconn`. The result - ing operation should be exactly the same as the example that used `Pipe()`. Under the covers, file descriptor passing involves creating a UNIX domain socket and the `sendmsg()` method of sockets. Since this technique is not widely known, here is a different implementation of the server that shows how to pass descriptors using sockets:

```
# server.py import socket
```

```
import struct
```

```
def send_fd(sock, fd):
```

```
    """ Send a single file descriptor. """ sock.sendmsg([b'x'],
```

```
                [(socket.SOL_SOCKET, socket.SCM_RIGHTS, struct.pack('i', fd))])
```

```
    ack = sock.recv(2) assert ack == b'OK'
```

```
def server(work_address, port):
```

```
    # Wait for the worker to connect work_serv = socket.socket(socket.AF_UNIX,
    socket.SOCK_STREAM) work_serv.bind(work_address) work_serv.listen(1) worker, addr
    = work_serv.accept()
```

```
    # Now run a TCP/IP server and send clients to worker s = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM) s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
    True) s.bind(('',port)) s.listen(1) while True:
```

```
        client, addr = s.accept() print('SERVER: Got connection from', addr) send_fd(worker,
        client.fileno()) client.close()
```

```
if __name__ == '__main__':
```

```
import sys if len(sys.argv) != 3:
    print('Usage: server.py server_address port', file=sys.stderr) raise SystemExit(1)

server(sys.argv[1], int(sys.argv[2]))
```

Here is an implementation of the worker using sockets:

```
# worker.py import socket import struct
```

```
def recv_fd(sock):
```

```
    """ Receive a single file descriptor """ msg, ancdata, flags, addr = sock.recvmsg(1,
        socket.CMSG_LEN(struct.calcsize('i')))

    msg_level, msg_type, msg_data = ancdata[0] assert msg_level ==
    socket.SOL_SOCKET and msg_type == socket.SCM_RIGHTS sock.sendall(b'OK')

    return struct.unpack('i', msg_data)[0]
```

```
def worker(server_address):
```

```
    serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    serv.connect(server_address) while True:

        fd = recv_fd(serv) print('WORKER: GOT FD', fd) with socket.socket(socket.AF_INET,
            socket.SOCK_STREAM, fileno=fd) as client:

            while True:
                msg = client.recv(1024) if not msg:
                    break

                print('WORKER: RECV {!r}'.format(msg)) client.send(msg)
```

```
if __name__ == '__main__':
```

```
    import sys if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr) raise SystemExit(1)

    worker(sys.argv[1])
```

If you are going to use file-descriptor passing in your program, it is advisable to read more about it in an advanced text, such as *Unix Network Programming* by W. Richard Stevens (Prentice Hall, 1990). Passing file descriptors on Windows uses a different technique than Unix (not shown). For that platform, it is advisable to study the source code to `multiprocessing.reduction` in close detail to see how it works.

11.12 理解事件驱动的IO

问题

You have heard about packages based on “event-driven” or “asynchronous” I/O, but you’re not entirely sure what it means, how it actually works under the covers, or how it might impact your program if you use it.

解决方案

At a fundamental level, event-driven I/O is a technique that takes basic I/O operations (e.g., reads and writes) and converts them into events that must be handled by your program. For example, whenever data was received on a socket, it turns into a “receive” event that is handled by some sort of callback method or function that you supply to respond to it. As a possible starting point, an event-driven framework might start with a base class that implements a series of basic event handler methods like this:

```
class EventHandler:
    def fileno(self):
        'Return the associated file descriptor' raise NotImplemented('must implement')

    def wants_to_receive(self):
        'Return True if receiving is allowed' return False

    def handle_receive(self):
        'Perform the receive operation' pass

    def wants_to_send(self):
        'Return True if sending is requested' return False

    def handle_send(self):
        'Send outgoing data' pass
```

Instances of this class then get plugged into an event loop that looks like this:

```
import select
```

```
def event_loop(handlers):
    while True:
        wants_recv = [h for h in handlers if h.wants_to_receive()]
        wants_send = [h for h in handlers if h.wants_to_send()]
        can_recv, can_send, _ = select.select(wants_recv,
        wants_send, [])
        for h in can_recv:
            h.handle_receive()

        for h in can_send:
            h.handle_send()
```

That's it! The key to the event loop is the `select()` call, which polls file descriptors for activity. Prior to calling `select()`, the event loop simply queries all of the handlers to see which ones want to receive or send. It then supplies the resulting lists to `select()`. As a result, `select()` returns the list of objects that are ready to receive or send. The corresponding `handle_receive()` or `handle_send()` methods are triggered. To write applications, specific instances of `EventHandler` classes are created. For example, here are two simple handlers that illustrate two UDP-based network services:

```
import socket
import time
```

```
class UDPServer(EventHandler):
```

```
    def __init__(self, address):
```

```
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(address)
```

```
    def fileno(self):
```

```
        return self.sock.fileno()
```

```
    def wants_to_receive(self):
```

```
        return True
```

```
class UDPTimeServer(UDPServer):
```

```
    def handle_receive(self):
```

```
        msg, addr = self.sock.recvfrom(1)
        self.sock.sendto(time.ctime().encode('ascii'), addr)
```

```
class UDPEchoServer(UDPServer):
```

```
    def handle_receive(self):
```

```
        msg, addr = self.sock.recvfrom(8192)
        self.sock.sendto(msg, addr)
```

```
if __name__ == '__main__':
```

```
    handlers = [ UDPTimeServer(('',14000)), UDPEchoServer(('',15000)) ]
```

```
    event_loop(handlers)
```

To test this code, you can try connecting to it from another Python interpreter:

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 14000))
0
>>> s.recvfrom(128)
(b'Tue Sep 18 14:29:23 2012', ('127.0.0.1', 14000))
>>> s.sendto(b'Hello', ('localhost', 15000))
5
>>> s.recvfrom(128)
(b'Hello', ('127.0.0.1', 15000))
>>>
```

Implementing a TCP server is somewhat more complex, since each client involves the instantiation of a new handler object. Here is an example of a TCP echo client.

```
class TCPServer(EventHandler):
    def __init__(self, address, client_handler, handler_list):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.sock.bind(address) self.sock.listen(1) self.client_handler = client_handler
        self.handler_list = handler_list

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

    def handle_receive(self):
        client, addr = self.sock.accept() # Add the client to the event loop's handler list
        self.handler_list.append(self.client_handler(client, self.handler_list))

class TCPClient(EventHandler):
    def __init__(self, sock, handler_list):
        self.sock = sock self.handler_list = handler_list self.outgoing = bytearray()

    def fileno(self):
        return self.sock.fileno()

    def close(self):
        self.sock.close() # Remove myself from the event loop's handler list
        self.handler_list.remove(self)

    def wants_to_send(self):
        return True if self.outgoing else False

    def handle_send(self):
        nsent = self.sock.send(self.outgoing) self.outgoing = self.outgoing[nsent:]

class TCPEchoClient(TCPClient):
    def wants_to_receive(self):
        return True

    def handle_receive(self):
        data = self.sock.recv(8192) if not data:

            self.close()

        else:
            self.outgoing.extend(data)

if __name__ == '__main__':
    handlers = [] handlers.append(TCPServer(('',16000), TCPEchoClient, handlers))
    event_loop(handlers)
```

The key to the TCP example is the addition and removal of clients from the handler list. On each connection, a new handler is created for the client and added to the list. When the connection is closed, each client must take care to remove themselves from the list. If you run this program and try connecting with Telnet or some similar tool, you'll see it echoing received data back to you. It should easily handle multiple clients.

讨论

Virtually all event-driven frameworks operate in a manner that is similar to that shown in the solution. The actual implementation details and overall software architecture might vary greatly, but at the core, there is a polling loop that checks sockets for activity and which performs operations in response. One potential benefit of event-driven I/O is that it can handle a very large number of simultaneous connections without ever using threads or processes. That is, the `select()` call (or equivalent) can be used to monitor hundreds or thousands of sockets and respond to events occurring on any of them. Events are handled one at a time by the event loop, without the need for any other concurrency primitives. The downside to event-driven I/O is that there is no true concurrency involved. If any of the event handler methods blocks or performs a long-running calculation, it blocks the progress of everything. There is also the problem of calling out to library functions that aren't written in an event-driven style. There is always the risk that some library call will block, causing the event loop to stall. Problems with blocking or long-running calculations can be solved by sending the work out to a separate thread or process. However, coordinating threads and processes with an event loop is tricky. Here is an example of code that will do it using the `concurrent.futures` module:

```
from concurrent.futures import ThreadPoolExecutor import os
```

```
class ThreadPoolHandler(EventHandler):
```

```
    def __init__(self, nworkers):
```

```
        if os.name == 'posix':
```

```
            self.signal_done_sock, self.done_sock = socket.socketpair()
```

```
        else:
```

```
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
            server.bind(('127.0.0.1', 0)) server.listen(1) self.signal_done_sock =
```

```
            socket.socket(socket.AF_INET,
```

```
                            socket.SOCK_STREAM)
```

```
            self.signal_done_sock.connect(server.getsockname()) self.done_sock, _ =
```

```
            server.accept() server.close()
```

```
        self.pending = [] self.pool = ThreadPoolExecutor(nworkers)
```

```

def fileno(self):
    return self.done_sock.fileno()

# Callback that executes when the thread is done def _complete(self, callback, r):
    self.pending.append((callback, r.result())) self.signal_done_sock.send(b'x')

# Run a function in a thread pool def run(self, func, args=(), kwargs={}, *, callback):
    r = self.pool.submit(func, *args, **kwargs) r.add_done_callback(lambda r:
    self._complete(callback, r))

def wants_to_receive(self):
    return True

# Run callback functions of completed work def handle_receive(self):
    # Invoke all pending callback functions for callback, result in self.pending:
        callback(result) self.done_sock.recv(1)

    self.pending = []

```

In this code, the `run()` method is used to submit work to the pool along with a callback function that should be triggered upon completion. The actual work is then submitted to a `ThreadPoolExecutor` instance. However, a really tricky problem concerns the coordination of the computed result and the event loop. To do this, a pair of sockets are created under the covers and used as a kind of signaling mechanism. When work is completed by the thread pool, it executes the `_complete()` method in the class. This method queues up the pending callback and result before writing a byte of data on one of these sockets. The `fileno()` method is programmed to return the other socket. Thus, when this byte is written, it will signal to the event loop that something has happened. The `handle_receive()` method, when triggered, will then execute all of the callback functions for previously submitted work. Frankly, it's enough to make one's head spin. Here is a simple server that shows how to use the thread pool to carry out a long-running calculation:

```

# A really bad Fibonacci implementation def fib(n):

    if n < 2:
        return 1

    else:
        return fib(n - 1) + fib(n - 2)

class UDPFibServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(128) n = int(msg) pool.run(fib, (n,), callback=lambda r:

```



```

self.respond(r, addr))

def respond(self, result, addr):
    self.sock.sendto(str(result).encode('ascii'), addr)

if __name__ == '__main__':
    pool = ThreadPoolHandler(16) handlers = [ pool, UDPFibServer(('',16000))]
    event_loop(handlers)

```

To try this server, simply run it and try some experiments with another Python program:

```

from socket import * sock = socket(AF_INET, SOCK_DGRAM) for x in range(40):

    sock.sendto(str(x).encode('ascii'), ('localhost', 16000)) resp = sock.recvfrom(8192)
    print(resp[0])

```

You should be able to run this program repeatedly from many different windows and have it operate without stalling other programs, even though it gets slower and slower as the numbers get larger. Having gone through this recipe, should you use its code? Probably not. Instead, you should look for a more fully developed framework that accomplishes the same task. However, if you understand the basic concepts presented here, you'll understand the core techniques used to make such frameworks operate. As an alternative to callback-based programming, event-driven code will sometimes use coroutines. See Recipe 12.12 for an example.

11.13 发送与接收大型数组

问题

You want to send and receive large arrays of contiguous data across a network connection, making as few copies of the data as possible.

解决方案

The following functions utilize memoryviews to send and receive large arrays:

```

# zerocopy.py

def send_from(arr, dest):
    view = memoryview(arr).cast('B') while len(view):

        nsent = dest.send(view) view = view[nsent:]

def recv_into(arr, source):

```

```
view = memoryview(arr).cast('B') while len(view):
```

```
    nrecv = source.recv_into(view) view = view[nrecv:]
```

To test the program, first create a server and client program connected over a socket. In the server:

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.bind(('', 25000))
>>> s.listen(1)
>>> c,a = s.accept()
>>>
```

In the client (in a separate interpreter):

```
>>> from socket import *
>>> c = socket(AF_INET, SOCK_STREAM)
>>> c.connect(('localhost', 25000))
>>>
```

Now, the whole idea of this recipe is that you can blast a huge array through the connection. In this case, arrays might be created by the array module or perhaps numpy. For example: # Server >>> import numpy >>> a = numpy.arange(0.0, 50000000.0) >>> send_from(a, c) >>>

```
# Client >>> import numpy >>> a = numpy.zeros(shape=50000000, dtype=float) >>>
a[0:10] array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]) >>> recv_into(a, c) >>> a[0:10] array([ 0., 1., 2., 3.,
4., 5., 6., 7., 8., 9.]) >>>
```

讨论

In data-intensive distributed computing and parallel programming applications, it's not uncommon to write programs that need to send/receive large chunks of data. However, to do this, you somehow need to reduce the data down to raw bytes for use with low-level network functions. You may also need to slice the data into chunks, since most network-related functions aren't able to send or receive huge blocks of data entirely all at once. One approach is to serialize the data in some way—possibly by converting into a byte string. However, this usually ends up making a copy of the data. Even if you do this piecemeal, your code still ends up making a lot of little copies.

This recipe gets around this by playing a sneaky trick with memoryviews. Essentially, a memoryview is an overlay of an existing array. Not only that, memoryviews can be cast to different types to allow interpretation of the data in a different manner. This is the purpose of the following statement: `view = memoryview(arr).cast('B')`

It takes an array `arr` and casts into a memoryview of unsigned bytes. In this form, the view can be passed to socket-related functions, such as `sock.send()` or `send.recv_into()`. Under the covers, those methods are able to work directly with the memory region. For example, `sock.send()` sends data directly from memory without a copy. `send.recv_into()` uses the memoryview as the input buffer for the receive operation. The remaining complication is the fact that the socket functions may only work with partial data. In general, it will take many different `send()` and `recv_into()` calls to transmit the entire array. Not to worry. After each operation, the view is sliced by the number of sent or received bytes to produce a new view. The new view is also a memory overlay. Thus, no copies are made. One issue here is that the receiver has to know in advance how much data will be sent so that it can either preallocate an array or verify that it can receive the data into an existing array. If this is a problem, the sender could always arrange to send the size first, followed by the array data.

第十二章：并发编程

对于并发编程, Python有多种长期支持的方法, 包括多线程, 调用子进程, 以及各种各样的关于生成器函数的技巧. 这一章将会给出并发编程各种方面的技巧, 包括通用的多线程技术以及并行计算的实现方法.

像经验丰富的程序员所知道的那样, 大家担心并发的程序有潜在的危险. 因此, 本章的主要目标之一是给出更加可信赖和易调试的代码.

Contents:

12.1 启动与停止线程

问题

你要为需要并发执行的代码创建/销毁线程

解决方案

`threading` 库可以在单独的线程中执行任何的在 Python 中可以调用的对象。你可以创建一个 `Thread` 对象并将你要执行的对象以 `target` 参数的形式提供给该对象。下面是一个简单的例子:

```
# Code to execute in an independent thread
import time
def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Create and launch a thread
from threading import Thread
t = Thread(target=countdown, args=(10,))
t.start()
```

当你创建好一个线程对象后，该对象并不会立即执行，除非你调用它的 `start()` 方法（当你调用 `start()` 方法时，它会调用你传递进来的函数，并把你传递进来的参数传递给该函数）。Python 中的线程会在一个单独的系统级线程中执行（比如说一个 POSIX 线程或者一个 Windows 线程），这些线程将由操作系统来全权管理。线程一旦启动，将独立执行直到目标函数返回。你可以查询一个线程对象的状态，看它是否还在执行：

```
if t.is_alive():
    print('Still running')
else:
    print('Completed')
```

你也可以将一个线程加入到当前线程，并等待它终止：

```
t.join()
```

Python 解释器在所有线程都终止后才继续执行代码剩余的部分。对于需要长时间运行的线程或者需要一直运行的后台任务，你应当考虑使用后台线程。例如：

```
t = Thread(target=countdown, args=(10,), daemon=True)
t.start()
```

后台线程无法等待，不过，这些线程会在主线程终止时自动销毁。除了如上所示的两个操作，并没有太多可以对线程做的事情。你无法结束一个线程，无法给它发送信号，无法调整它的调度，也无法执行其他高级操作。如果需要这些特性，你需要自己添加。比如说，如果你需要终止线程，那么这个线程必须通过编程在某个特定点轮询来退出。你可以像下边这样把线程放入一个类中：

```

class CountdownTask:
    def __init__(self):
        self._running = True

    def terminate(self):
        self._running = False

    def run(self, n):
        while self._running and n > 0:
            print('T-minus', n)
            n -= 1
            time.sleep(5)

c = CountdownTask()
t = Thread(target=c.run, args=(10,))
t.start()
c.terminate() # Signal termination
t.join()      # Wait for actual termination (if needed)

```

如果线程执行一些像I/O这样的阻塞操作，那么通过轮询来终止线程将使得线程之间的协调变得非常棘手。比如，如果一个线程一直阻塞在一个I/O操作上，它就永远无法返回，也就无法检查自己是否已经被结束了。要正确处理这些问题，你需要利用超时循环来小心操作线程。例子如下：

```

class IOTask:
    def terminate(self):
        self._running = False

    def run(self, sock):
        # sock is a socket
        sock.settimeout(5)          # Set timeout period
        while self._running:
            # Perform a blocking I/O operation w/ timeout
            try:
                data = sock.recv(8192)
                break
            except socket.timeout:
                continue
            # Continued processing
            ...
        # Terminated
        return

```

讨论

由于全局解释锁（GIL）的原因，Python 的线程被限制到同一时刻只允许一个线程执行这样一个执行模型。所以，Python 的线程更适用于处理 I/O 和其他需要并发执行的阻塞操作（比如等待 I/O、等待从数据库获取数据等等），而不是需要多处理器并行的计算密集型任务。

有时你会看到下边这种通过继承 `Thread` 类来实现的线程：

```
from threading import Thread

class CountdownThread(Thread):
    def __init__(self, n):
        super().__init__()
        self.n = n
    def run(self):
        while self.n > 0:

            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

c = CountdownThread(5)
c.start()
```

尽管这样也可以工作，但这使得你的代码依赖于 `threading` 库，所以你的这些代码只能在线程上下文中使用。上文所写的那些代码、函数都是与 `threading` 库无关的，这样就使得这些代码可以被用在其他的上下文中，可能与线程有关，也可能与线程无关。比如，你可以通过 `multiprocessing` 模块在一个单独的进程中执行你的代码：

```
import multiprocessing
c = CountdownTask(5)
p = multiprocessing.Process(target=c.run)
p.start()
```

再次重申，这段代码仅适用于 `CountdownTask` 类是以独立于实际的并发手段（多线程、多进程等等）实现的情况。

12.2 判断线程是否已经启动

问题

你已经启动了一个线程，但是你知道它是不是真的已经开始运行了。

解决方案

线程的一个关键特性是每个线程都是独立运行且状态不可预测。如果程序中的其他线程需要通过判断某个线程的状态来确定自己下一步的操作，这时线程同步问题就会变得非常棘手。为了解决这些问题，我们需要使用 `threading` 库中的 `Event` 对象。`Event` 对象包含一个可由线程设置的信号标志，它允许线程等待某些事件的发生。在初始情况下，`event` 对象中的信号标志被设置为假。如果有线程等待一个 `event` 对象，而这个 `event` 对象的标志为假，那么这个线程将会被一直阻塞直至该标志为真。一个线程如果将一个 `event` 对象的信号标志设置为真，它将唤醒所有等待这个 `event` 对象的线程。如果一个线程等待一个已经被设置为真的 `event` 对象，那么它将忽略这个事件，继续执行。下边的代码展示了如何使用 `Event` 来协调线程的启动：

```
from threading import Thread, Event
import time

# Code to execute in an independent thread
def countdown(n, started_evt):
    print('countdown starting')
    started_evt.set()
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Create the event object that will be used to signal startup
started_evt = Event()

# Launch the thread and pass the startup event
print('Launching countdown')
t = Thread(target=countdown, args=(10,started_evt))
t.start()

# Wait for the thread to start
started_evt.wait()
print('countdown is running')
```

当你执行这段代码，“countdown is running”总是显示在“countdown starting”之后显示。这是由于使用 `event` 来协调线程，使得主线程要等到 `countdown()` 函数输出启动信息后，才能继续执行。

讨论

`event` 对象最好单次使用，就是说，你创建一个 `event` 对象，让某个线程等待这个对象，一旦这个对象被设置为真，你就应该丢弃它。尽管可以通过 `clear()` 方法来重置 `event` 对象，但是很难确保安全地清理 `event` 对象并对它重新赋值。很可能发生错过事件、死锁或者其他问题（特别是，你无法保证重置 `event` 对象的代码会在线程再次等待这个 `event`

对象之前执行)。如果一个线程需要不停地重复使用 `event` 对象，你最好使用 `Condition` 对象来代替。下面的代码使用 `Condition` 对象实现了一个周期定时器，每当定时器超时的时候，其他线程都可以监测到：


```

import threading
import time

class PeriodicTimer:
    def __init__(self, interval):
        self._interval = interval
        self._flag = 0
        self._cv = threading.Condition()

    def start(self):
        t = threading.Thread(target=self.run)
        t.daemon = True

        t.start()

    def run(self):
        """
        Run the timer and notify waiting threads after each interval
        """
        while True:
            time.sleep(self._interval)
            with self._cv:
                self._flag ^= 1
                self._cv.notify_all()

    def wait_for_tick(self):
        """
        Wait for the next tick of the timer
        """
        with self._cv:
            last_flag = self._flag
            while last_flag == self._flag:
                self._cv.wait()

# Example use of the timer
ptimer = PeriodicTimer(5)
ptimer.start()

# Two threads that synchronize on the timer
def countdown(nticks):
    while nticks > 0:
        ptimer.wait_for_tick()
        print('T-minus', nticks)
        nticks -= 1

def countup(last):
    n = 0
    while n < last:
        ptimer.wait_for_tick()
        print('Counting', n)
        n += 1

threading.Thread(target=countdown, args=(10,)).start()
threading.Thread(target=countup, args=(5,)).start()

```

event对象的一个重要特点是当它被设置为真时会唤醒所有等待它的线程。如果你只想唤醒单个线程，最好是使用信号量或者 `Condition` 对象来替代。考虑一下这段使用信号量实现的代码：

```
# Worker thread
def worker(n, sema):
    # Wait to be signaled
    sema.acquire()

    # Do some work
    print('Working', n)

# Create some threads
sema = threading.Semaphore(0)
nworkers = 10
for n in range(nworkers):
    t = threading.Thread(target=worker, args=(n, sema,))
    t.start()
```

运行上边的代码将会启动一个线程池，但是并没有什么事情发生。这是因为所有的线程都在等待获取信号量。每次信号量被释放，只有一个线程会被唤醒并执行，示例如下：

```
>>> sema.release()
Working 0
>>> sema.release()
Working 1
>>>
```

编写涉及到大量的线程间同步问题的代码会让你痛不欲生。比较合适的方式是使用队列来进行线程间通信或者每个把线程当作一个Actor，利用Actor模型来控制并发。下一节将会介绍到队列，而Actor模型将在12.10节介绍。

12.3 线程间通信

问题

你的程序中有多个线程，你需要在这些线程之间安全地交换信息或数据

解决方案

从一个线程向另一个线程发送数据最安全的方式可能就是使用 `queue` 库中的队列了。创建一个被多个线程共享的 `Queue` 对象，这些线程通过使用 `put()` 和 `get()` 操作来向队列中添加或者删除元素。例如：

Queue 对象已经包含了必要的锁，所以您可以通过它在多个线程间多安全地共享数据。当使用队列时，协调生产者和消费者的关闭问题可能会有一些麻烦。一个通用的解决方法是在队列中放置一个特殊的只，当消费者读到这个值的时候，终止执行。例如：

```
from queue import Queue
from threading import Thread

# Object that signals shutdown
_sentinel = object()

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

    # Put the sentinel on the queue to indicate completion
    out_q.put(_sentinel)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Check for termination
        if data is _sentinel:
            in_q.put(_sentinel)
            break

        # Process the data
        ...
```

本例中有一个特殊的地方：消费者在读到这个特殊值之后立即又把它放回到队列中，将之传递下去。这样，所有监听这个队列的消费者线程就可以全部关闭了。尽管队列是最常见的线程间通信机制，但是仍然可以自己通过创建自己的数据结构并添加所需的锁和同步机制来实现线程间通信。最常见的方法是使用 **Condition** 变量来包装你的数据结构。下边这个例子演示了如何创建一个线程安全的优先级队列，如同1.5节中介绍的那样。

```
import heapq
import threading

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._count = 0
        self._cv = threading.Condition()
    def put(self, item, priority):
        with self._cv:
            heapq.heappush(self._queue, (-priority, self._count, item))
            self._count += 1
            self._cv.notify()

    def get(self):
        with self._cv:
            while len(self._queue) == 0:
                self._cv.wait()
            return heapq.heappop(self._queue)[-1]
```

使用队列来进行线程间通信是一个单向、不确定的过程。通常情况下，你没有办法知道接收数据的线程是什么时候接收到的数据并开始工作的。不过队列对象提供一些基本完成的特性，比如下边这个例子中的 `task_done()` 和 `join()`：

```

from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Process the data
        ...
        # Indicate completion
        in_q.task_done()

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

# Wait for all produced items to be consumed
q.join()

```

如果一个线程需要在一个“消费者”线程处理完特定的数据项时立即得到通知，你可以把要发送的数据和一个 `Event` 放到一起使用，这样“生产者”就可以通过这个 `Event` 对象来监测处理的过程了。示例如下：

```
from queue import Queue
from threading import Thread, Event

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        # Make an (data, event) pair and hand it to the consumer
        evt = Event()
        out_q.put((data, evt))
        ...
        # Wait for the consumer to process the item
        evt.wait()

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data, evt = in_q.get()
        # Process the data
        ...
        # Indicate completion
        evt.set()
```

讨论

基于简单队列编写多线程程序在多数情况下是一个比较明智的选择。从线程安全队列的底层实现来看，你无需在你的代码中使用锁和其他底层的同步机制，这些只会把你的程序弄得乱七八糟。此外，使用队列这种基于消息的通信机制可以被扩展到更大的应用范畴，比如，你可以把你的程序放入多个进程甚至是分布式系统而无需改变底层的队列结构。使用线程队列有一个要注意的问题是，向队列中添加数据项时并不会复制此数据项，线程间通信实际上是在线程间传递对象引用。如果你担心对象的共享状态，那你最好只传递不可修改的数据结构（如：整型、字符串或者元组）或者一个对象的深拷贝。例如：

```

from queue import Queue
from threading import Thread
import copy

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(copy.deepcopy(data))

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()
        # Process the data
        ...

```

`Queue` 对象提供一些在当前上下文很有用的附加特性。比如在创建 `Queue` 对象时提供可选的 `size` 参数来限制可以添加到队列中的元素数量。对于“生产者”与“消费者”速度有差异的情况，为队列中的元素数量添加上限是有意义的。比如，一个“生产者”产生项目的速度比“消费者”“消费”的速度快，那么使用固定大小的队列就可以在队列已满的时候阻塞队列，以免未预期的连锁效应扩散整个程序造成死锁或者程序运行失常。在通信的线程之间进行“流量控制”是一个看起来容易实现起来困难的问题。如果你发现自己曾经试图通过摆弄队列大小来解决一个问题，这也许就标志着你的程序可能存在脆弱设计或者固有的可伸缩问题。 `get()` 和 `put()` 方法都支持非阻塞方式和设定超时，例如：

```

import queue
q = queue.Queue()

try:
    data = q.get(block=False)
except queue.Empty:
    ...

try:
    q.put(item, block=False)
except queue.Full:
    ...

try:
    data = q.get(timeout=5.0)
except queue.Empty:
    ...

```

这些操作都可以用来避免当执行某些特定队列操作时发生无限阻塞的情况，比如，一个非阻塞的 `put()` 方法和一个固定大小的队列一起使用，这样当队列已满时就可以执行不同的代码。比如输出一条日志信息并丢弃。

```
def producer(q):
    ...
    try:
        q.put(item, block=False)
    except queue.Full:
        log.warning('queued item %r discarded!', item)
```

如果你试图让消费者线程在执行像 `q.get()` 这样的操作时，超时自动终止以便检查终止标志，你应该使用 `q.get()` 的可选参数 `timeout`，如下：

```
_running = True

def consumer(q):
    while _running:
        try:
            item = q.get(timeout=5.0)
            # Process item
            ...
        except queue.Empty:
            pass
```

最后，有 `q.qsize()`，`q.full()`，`q.empty()` 等实用方法可以获取一个队列的当前大小和状态。但要注意，这些方法都不是线程安全的。可能你对一个队列使用 `empty()` 判断出这个队列为空，但同时另外一个线程可能已经向这个队列中插入一个数据项。所以，你最好不要在你的代码中使用这些方法。

12.4 给关键部分加锁

问题

你需要对多线程程序中的临界区加锁以避免竞争条件。

解决方案

要在多线程程序中安全使用可变对象，你需要使用 `threading` 库中的 `Lock` 对象，就像下边这个例子这样：


```
import threading

class SharedCounter:
    """
    A counter object that can be shared by multiple threads.
    """
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self, delta=1):
        """
        Increment the counter with locking
        """
        with self._value_lock:
            self._value += delta

    def decr(self, delta=1):
        """
        Decrement the counter with locking
        """
        with self._value_lock:
            self._value -= delta
```

`Lock` 对象和 `with` 语句块一起使用可以保证互斥执行，就是每次只有一个线程可以执行 `with` 语句包含的代码块。`with` 语句会在这个代码块执行前自动获取锁，在执行结束后自动释放锁。

讨论

线程调度本质上是不确定的，因此，在多线程程序中错误地使用锁机制可能会导致随机数据损坏或者其他的异常行为，我们称之为竞争条件。为了避免竞争条件，最好只在临界区（对临界资源进行操作的那部分代码）使用锁。在一些“老的” Python 代码中，显式获取和释放锁是很常见的。下边是一个上一个例子的变种：

```

import threading

class SharedCounter:
    """
    A counter object that can be shared by multiple threads.
    """
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self, delta=1):
        """
        Increment the counter with locking
        """
        self._value_lock.acquire()
        self._value += delta
        self._value_lock.release()

    def decr(self, delta=1):
        """
        Decrement the counter with locking
        """
        self._value_lock.acquire()
        self._value -= delta
        self._value_lock.release()

```

相比于这种显式调用的方法，`with` 语句更加优雅，也更不容易出错，特别是程序员可能会忘记调用 `release()` 方法或者程序在获得锁之后产生异常这两种情况（使用 `with` 语句可以保证在这两种情况下仍能正确释放锁）。为了避免出现死锁的情况，使用锁机制的程序应该设定为每个线程一次只允许获取一个锁。如果不能这样做的话，你就需要更高级的死锁避免机制，我们将在12.5节介绍。在 `threading` 库中还提供了其他的同步原语，比如 `RLock` 和 `Semaphore` 对象。但是根据以往经验，这些原语是用于一些特殊的情况，如果你只是需要简单地对可变对象进行锁定，那就不应该使用它们。一个 `RLock`（可重入锁）可以被同一个线程多次获取，主要用来实现基于监测对象模式的锁定和同步。在使用这种锁的情况下，当锁被持有时，只有一个线程可以使用完整的函数或者类中的方法。比如，你可以实现一个这样的 `SharedCounter` 类：

```

import threading

class SharedCounter:
    """
    A counter object that can be shared by multiple threads.
    """
    _lock = threading.RLock()
    def __init__(self, initial_value = 0):
        self._value = initial_value

    def incr(self, delta=1):
        """
        Increment the counter with locking
        """
        with SharedCounter._lock:
            self._value += delta

    def decr(self, delta=1):
        """
        Decrement the counter with locking
        """
        with SharedCounter._lock:
            self.incr(-delta)

```

在上边这个例子中，没有对每一个实例中的可变对象加锁，取而代之的是一个被所有实例共享的类级锁。这个锁用来同步类方法，具体来说就是，这个锁可以保证一次只有一个线程可以调用这个类方法。不过，与一个标准的锁不同的是，已经持有这个锁的方法在调用同样使用这个锁的方法时，无需再次获取锁。比如 `decr` 方法。这种实现方式的一个特点是，无论这个类有多少个实例都只用一个锁。因此在需要大量使用计数器的情况下内存效率更高。不过这样做也有缺点，就是在程序中使用大量线程并频繁更新计数器时会有争用锁的问题。信号量对象是一个建立在共享计数器基础上的同步原语。如果计数器不为0，`with` 语句将计数器减1，线程被允许执行。`with` 语句执行结束后，计数器加1。如果计数器为0，线程将被阻塞，直到其他线程结束将计数器加1。尽管你可以在程序中像标准锁一样使用信号量来做线程同步，但是这种方式并不被推荐，因为使用信号量为程序增加的复杂性会影响程序性能。相对于简单地作为锁使用，信号量更适用于那些需要在线程之间引入信号或者限制的程序。比如，你需要限制一段代码的并发访问量，你就可以像下面这样使用信号量完成：

```

from threading import Semaphore
import urllib.request

# At most, five threads allowed to run at once
_fetch_url_sema = Semaphore(5)

def fetch_url(url):
    with _fetch_url_sema:
        return urllib.request.urlopen(url)

```

如果你对线程同步原语的底层理论和实现感兴趣，可以参考操作系统相关书籍，绝大多数都有提及。

12.5 防止死锁的加锁机制

问题

You're writing a multithreaded program where threads need to acquire more than one lock at a time while avoiding deadlock.

解决方案

In multithreaded programs, a common source of deadlock is due to threads that attempt to acquire multiple locks at once. For instance, if a thread acquires the first lock, but then blocks trying to acquire the second lock, that thread can potentially block the progress of other threads and make the program freeze. One solution to deadlock avoidance is to assign each lock in the program a unique number, and to enforce an ordering rule that only allows multiple locks to be acquired in ascending order. This is surprisingly easy to implement using a context manager as follows:

```
import threading from contextlib import contextmanager
```

```
# Thread-local state to stored information on locks already acquired _local =  
threading.local()
```

```
@contextmanager def acquire(*locks):
```

```

# Sort locks by object identifier locks = sorted(locks, key=lambda x: id(x))

# Make sure lock order of previously acquired locks is not violated acquired =
getattr(_local, 'acquired', []) if acquired and max(id(lock) for lock in acquired) >=
id(locks[0]):

    raise RuntimeError("Lock Order Violation")

# Acquire all of the locks acquired.extend(locks) _local.acquired = acquired

try:
    for lock in locks:
        lock.acquire()

    yield

finally:
    # Release locks in reverse order of acquisition for lock in reversed(locks):

        lock.release()

    del acquired[-len(locks):]

```

To use this context manager, you simply allocate lock objects in the normal way, but use the `acquire()` function whenever you want to work with one or more locks. For example:

```

import threading x_lock = threading.Lock() y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock, y_lock):
            print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock, x_lock):
            print('Thread-2')

t1 = threading.Thread(target=thread_1) t1.daemon = True t1.start()

t2 = threading.Thread(target=thread_2) t2.daemon = True t2.start()

```

If you run this program, you'll find that it happily runs forever without deadlock—even though the acquisition of locks is specified in a different order in each function. The key to this recipe lies in the first statement that sorts the locks according to object identifier. By sorting the locks, they always get acquired in a consistent order regardless of how the user

might have provided them to acquire()). The solution uses thread-local storage to solve a subtle problem with detecting potential deadlock if multiple acquire() operations are nested. For example, suppose you wrote the code like this:

```
import threading
x_lock = threading.Lock()
y_lock = threading.Lock()
```

```
def thread_1():
```

```
    while True:
```

```
        with acquire(x_lock):
```

```
            with acquire(y_lock):
```

```
                print('Thread-1')
```

```
def thread_2():
```

```
    while True:
```

```
        with acquire(y_lock):
```

```
            with acquire(x_lock):
```

```
                print('Thread-2')
```

```
t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()
```

```
t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

If you run this version of the program, one of the threads will crash with an exception such as this:

Exception in thread Thread-1: Traceback (most recent call last):

```
File "/usr/local/lib/python3.3/threading.py", line 639, in _bootstrap_inner
    self.run()
```

```
File "/usr/local/lib/python3.3/threading.py", line 596, in run
    self._target(*self._args, **self._kwargs)
```

```
File "deadlock.py", line 49, in thread_1
    with acquire(y_lock):
```

```
File "/usr/local/lib/python3.3/contextlib.py", line 48, in __enter__
    return next(self.gen)
```

```
File "deadlock.py", line 15, in acquire
    raise RuntimeError("Lock Order Violation")
```

```
RuntimeError: Lock Order Violation >>>
```

This crash is caused by the fact that each thread remembers the locks it has already acquired. The `acquire()` function checks the list of previously acquired locks and enforces the ordering constraint that previously acquired locks must have an object ID that is less than the new locks being acquired.

讨论

The issue of deadlock is a well-known problem with programs involving threads (as well as a common subject in textbooks on operating systems). As a rule of thumb, as long as you can ensure that threads can hold only one lock at a time, your program will be deadlock free. However, once multiple locks are being acquired at the same time, all bets are off.

Detecting and recovering from deadlock is an extremely tricky problem with few elegant solutions. For example, a common deadlock detection and recovery scheme involves the use of a watchdog timer. As threads run, they periodically reset the timer, and as long as everything is running smoothly, all is well. However, if the program deadlocks, the watchdog timer will eventually expire. At that point, the program “recovers” by killing and then restarting itself. Deadlock avoidance is a different strategy where locking operations are carried out in a manner that simply does not allow the program to enter a deadlocked state. The solution in which locks are always acquired in strict order of ascending object ID can be mathematically proven to avoid deadlock, although the proof is left as an exercise to the reader (the gist of it is that by acquiring locks in a purely increasing order, you can’t get cyclic locking dependencies, which are a necessary condition for deadlock to occur). As a final example, a classic thread deadlock problem is the so-called “dining philosopher’s problem.” In this problem, five philosophers sit around a table on which there are five bowls of rice and five chopsticks. Each philosopher represents an independent thread and each chopstick represents a lock. In the problem, philosophers either sit and think or they eat rice. However, in order to eat rice, a philosopher needs two chopsticks. Unfortunately, if all of the philosophers reach over and grab the chopstick to their left, they’ll all just sit there with one stick and eventually starve to death. It’s a gruesome scene. Using the solution, here is a simple deadlock free implementation of the dining philosopher’s problem:

```
import threading
```

```
# The philosopher thread def philosopher(left, right):
```

```
    while True:
```

```
        with acquire(left,right):
```

```
            print(threading.currentThread(), 'eating')
```

```
# The chopsticks (represented by locks) NSTICKS = 5 chopsticks = [threading.Lock() for n in range(NSTICKS)]
```

```
# Create all of the philosophers for n in range(NSTICKS):
```

```
    t = threading.Thread(target=philosopher,  
                        args=(chopsticks[n], chopsticks[(n+1) % NSTICKS]))
```

```
    t.start()
```

Last, but not least, it should be noted that in order to avoid deadlock, all locking operations must be carried out using our `acquire()` function. If some fragment of code decided to acquire a lock directly, then the deadlock avoidance algorithm wouldn't work.

12.6 保存线程的状态信息

问题

You need to store state that's specific to the currently executing thread and not visible to other threads.

解决方案

Sometimes in multithreaded programs, you need to store data that is only specific to the currently executing thread. To do this, create a thread-local storage object using `threading.local()`. Attributes stored and read on this object are only visible to the executing thread and no others. As an interesting practical example of using thread-local storage, consider the `LazyConnection` context-manager class that was first defined in Recipe 8.3. Here is a slightly modified version that safely works with multiple threads:

```
from socket import socket, AF_INET, SOCK_STREAM import threading
```

```
class LazyConnection:
```

```
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):  
        self.address = address self.family = AF_INET self.type = SOCK_STREAM self.local =  
        threading.local()
```

```
    def __enter__(self):  
        if hasattr(self.local, 'sock'):  
            raise RuntimeError('Already connected')
```

```
        self.local.sock = socket(self.family, self.type) self.local.sock.connect(self.address)  
        return self.local.sock
```



```
def __exit__(self, exc_ty, exc_val, tb):
    self.local.sock.close() del self.local.sock
```

In this code, carefully observe the use of the `self.local` attribute. It is initialized as an instance of `threading.local()`. The other methods then manipulate a socket that's stored as `self.local.sock`. This is enough to make it possible to safely use an instance of `LazyConnection` in multiple threads. For example:

```
from functools import partial def test(conn):
```

```
    with conn as s:
```

```
        s.send(b'GET /index.html HTTP/1.0\r\n') s.send(b'Host: www.python.org\r\n')
```

```
        s.send(b'\r\n') resp = b''.join(iter(partial(s.recv, 8192), b''))
```

```
    print('Got {} bytes'.format(len(resp)))
```

```
if __name__ == '__main__':
```

```
    conn = LazyConnection(('www.python.org', 80))
```

```
    t1 = threading.Thread(target=test, args=(conn,)) t2 = threading.Thread(target=test,
    args=(conn,)) t1.start() t2.start() t1.join() t2.join()
```

The reason it works is that each thread actually creates its own dedicated socket connection (stored as `self.local.sock`). Thus, when the different threads perform socket operations, they don't interfere with one another as they are being performed on different sockets.

讨论

Creating and manipulating thread-specific state is not a problem that often arises in most programs. However, when it does, it commonly involves situations where an object being used by multiple threads needs to manipulate some kind of dedicated system resource, such as a socket or file. You can't just have a single socket object shared by everyone because chaos would ensue if multiple threads ever started reading and writing on it at the same time. Thread-local storage fixes this by making such resources only visible in the thread where they're being used. In this recipe, the use of `threading.local()` makes the `LazyConnection` class support one connection per thread, as opposed to one connection for the entire process. It's a subtle but interesting distinction. Under the covers, an instance of `threading.local()` maintains a separate instance dictionary for each thread. All of the usual instance operations of getting, setting, and deleting values just manipulate the per-thread dictionary. The fact that each thread uses a separate dictionary is what provides the isolation of data.

12.7 创建一个线程池

问题

You want to create a pool of worker threads for serving clients or performing other kinds of work.

解决方案

The `concurrent.futures` library has a `ThreadPoolExecutor` class that can be used for this purpose. Here is an example of a simple TCP server that uses a thread-pool to serve clients:

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor
```

```
def echo_client(sock, client_addr):
```

```
    """ Handle a client connection """ print('Got connection from', client_addr) while True:
```

```
        msg = sock.recv(65536) if not msg:
```

```
            break
```

```
        sock.sendall(msg)
```

```
    print('Client closed connection') sock.close()
```

```
def echo_server(addr):
```

```
    pool = ThreadPoolExecutor(128) sock = socket(AF_INET, SOCK_STREAM)
```

```
    sock.bind(addr) sock.listen(5) while True:
```

```
        client_sock, client_addr = sock.accept() pool.submit(echo_client, client_sock,
        client_addr)
```

```
echo_server(('',15000))
```

If you want to manually create your own thread pool, it's usually easy enough to do it using a `Queue`. Here is a slightly different, but manual implementation of the same code:

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue
```

```
def echo_client(q):
```

```
    """ Handle a client connection """ sock, client_addr = q.get() print('Got connection from',
    client_addr) while True:
```

```
msg = sock.recv(65536) if not msg:
```

```
    break
```

```
sock.sendall(msg)
```

```
print('Client closed connection')
```

```
sock.close()
```

```
def echo_server(addr, nworkers):
```

```
    # Launch the client workers q = Queue() for n in range(nworkers):
```

```
        t = Thread(target=echo_client, args=(q,)) t.daemon = True t.start()
```

```
    # Run the server sock = socket(AF_INET, SOCK_STREAM) sock.bind(addr) sock.listen(5)
```

```
    while True:
```

```
        client_sock, client_addr = sock.accept() q.put((client_sock, client_addr))
```

```
echo_server(('',15000), 128)
```

One advantage of using `ThreadPoolExecutor` over a manual implementation is that it makes it easier for the submitter to receive results from the called function. For example, you could write code like this:

```
from concurrent.futures import ThreadPoolExecutor import urllib.request
```

```
def fetch_url(url):
```

```
    u = urllib.request.urlopen(url) data = u.read() return data
```

```
pool = ThreadPoolExecutor(10) # Submit work to the pool a = pool.submit(fetch_url,  
'http://www.python.org') b = pool.submit(fetch_url, 'http://www.pypy.org')
```

```
# Get the results back x = a.result() y = b.result()
```

The result objects in the example handle all of the blocking and coordination needed to get data back from the worker thread. Specifically, the operation `a.result()` blocks until the corresponding function has been executed by the pool and returned a value.

讨论

Generally, you should avoid writing programs that allow unlimited growth in the number of threads. For example, take a look at the following server:

```
from threading import Thread from socket import socket, AF_INET, SOCK_STREAM
```

```
def echo_client(sock, client_addr):
```

```
    """ Handle a client connection """ print('Got connection from', client_addr) while True:
```

```
        msg = sock.recv(65536) if not msg:
```

```
            break
```

```
        sock.sendall(msg)
```

```
    print('Client closed connection') sock.close()
```

```
def echo_server(addr, nworkers):
```

```
    # Run the server sock = socket(AF_INET, SOCK_STREAM) sock.bind(addr) sock.listen(5)
```

```
    while True:
```

```
        client_sock, client_addr = sock.accept() t = Thread(target=echo_client, args=
        (client_sock, client_addr)) t.daemon = True t.start()
```

```
echo_server(('',15000))
```

Although this works, it doesn't prevent some asynchronous hipster from launching an attack on the server that makes it create so many threads that your program runs out of resources and crashes (thus further demonstrating the "evils" of using threads). By using a pre-initialized thread pool, you can carefully put an upper limit on the amount of supported concurrency. You might be concerned with the effect of creating a large number of threads. However, modern systems should have no trouble creating pools of a few thousand threads. Moreover, having a thousand threads just sitting around waiting for work isn't going to have much, if any, impact on the performance of other code (a sleeping thread does just that—nothing at all). Of course, if all of those threads wake up at the same time and start hammering on the CPU, that's a different story—especially in light of the Global Interpreter Lock (GIL). Generally, you only want to use thread pools for I/O-bound processing. One possible concern with creating large thread pools might be memory use. For example, if you create 2,000 threads on OS X, the system shows the Python process using up more than 9 GB of virtual memory. However, this is actually somewhat misleading. When creating a thread, the operating system reserves a region of virtual memory to hold the thread's execution stack (often as large as 8 MB). Only a small fragment of this memory is actually mapped to real memory, though. Thus, if you look a bit closer, you might find the Python process is using far less real memory (e.g., for 2,000 threads, only

70 MB of real memory is used, not 9 GB). If the size of the virtual memory is a concern, you can dial it down using the `threading.stack_size()` function. For example:

```
import threading threading.stack_size(65536)
```

If you add this call and repeat the experiment of creating 2,000 threads, you'll find that the Python process is now only using about 210 MB of virtual memory, although the amount of real memory in use remains about the same. Note that the thread stack size must be at least 32,768 bytes, and is usually restricted to be a multiple of the system memory page size (4096, 8192, etc.).

12.8 简单的并行编程

问题

You have a program that performs a lot of CPU-intensive work, and you want to make it run faster by having it take advantage of multiple CPUs.

解决方案

The `concurrent.futures` library provides a `ProcessPoolExecutor` class that can be used to execute computationally intensive functions in a separately running instance of the Python interpreter. However, in order to use it, you first need to have some computationally intensive work. Let's illustrate with a simple yet practical example. Suppose you have an entire directory of gzip-compressed Apache web server logs:

```
logs/
  20120701.log.gz 20120702.log.gz 20120703.log.gz 20120704.log.gz 20120705.log.gz
  20120706.log.gz ...
```

Further suppose each log file contains lines like this:

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71 210.212.209.67 -
- [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875 210.212.209.67 - -
[10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369 61.135.216.105 - -
[10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 - ...
```

Here is a simple script that takes this data and identifies all hosts that have accessed the `robots.txt` file:

```
# findrobots.py
```

```
import gzip import io import glob
```

```
def find_robots(filename):
```

```
    """ Find all of the hosts that access robots.txt in a single log file """ robots = set() with
    gzip.open(filename) as f:
```

```

    for line in io.TextIOWrapper(f,encoding='ascii'):
        fields = line.split() if fields[6] == '/robots.txt':

            robots.add(fields[0])

return robots

def find_all_robots(logdir):
    """ Find all hosts across and entire sequence of files """ files = glob.glob(logdir+'/*.log.gz')
    all_robots = set() for robots in map(find_robots, files):

        all_robots.update(robots)

return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs') for ipaddr in robots:

        print(ipaddr)

```

The preceding program is written in the commonly used map-reduce style. The function `find_robots()` is mapped across a collection of filenames and the results are combined into a single result (the `all_robots` set in the `find_all_robots()` function). Now, suppose you want to modify this program to use multiple CPUs. It turns out to be easy—simply replace the `map()` operation with a similar operation carried out on a process pool from the `concurrent.futures` library. Here is a slightly modified version of the code:

```
# findrobots.py
```

```
import gzip import io import glob from concurrent import futures
```

```

def find_robots(filename):
    """ Find all of the hosts that access robots.txt in a single log file

    """ robots = set() with gzip.open(filename) as f:

        for line in io.TextIOWrapper(f,encoding='ascii'):
            fields = line.split() if fields[6] == '/robots.txt':

                robots.add(fields[0])

return robots

def find_all_robots(logdir):
    """ Find all hosts across and entire sequence of files """ files = glob.glob(logdir+'/*.log.gz')
    all_robots = set() with futures.ProcessPoolExecutor() as pool:

        for robots in pool.map(find_robots, files):
            all_robots.update(robots)

```

```

return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs') for ipaddr in robots:

        print(ipaddr)

```

With this modification, the script produces the same result but runs about 3.5 times faster on our quad-core machine. The actual performance will vary according to the number of CPUs available on your machine.

讨论

Typical usage of a `ProcessPoolExecutor` is as follows: `from concurrent.futures import ProcessPoolExecutor`

with `ProcessPoolExecutor()` as pool:

...do work in parallel using pool ...

Under the covers, a `ProcessPoolExecutor` creates `N` independent running Python interpreters where `N` is the number of available CPUs detected on the system. You can change the number of processes created by supplying an optional argument to `ProcessPoolExecutor(N)`. The pool runs until the last statement in the `with` block is executed, at which point the process pool is shut down. However, the program will wait until all submitted work has been processed. Work to be submitted to a pool must be defined in a function. There are two methods for submission. If you are trying to parallelize a list comprehension or a `map()` operation, you use `pool.map()`:

```
# A function that performs a lot of work
def work(x):
```

```
    ... return result
```

```
# Nonparallel code
results = map(work, data)
```

```
# Parallel implementation with ProcessPoolExecutor() as pool:
```

```
    results = pool.map(work, data)
```

Alternatively, you can manually submit single tasks using the `pool.submit()` method:

```
# Some function
def work(x):
```

```
    ... return result
```

with ProcessPoolExecutor() as pool:

```
... # Example of submitting work to the pool future_result = pool.submit(work, arg)
```

```
# Obtaining the result (blocks until done) r = future_result.result() ...
```

If you manually submit a job, the result is an instance of Future. To obtain the actual result, you call its result() method. This blocks until the result is computed and returned by the pool. Instead of blocking, you can also arrange to have a callback function triggered upon completion instead. For example:

```
def when_done(r):
```

```
    print('Got:', r.result())
```

with ProcessPoolExecutor() as pool:

```
    future_result = pool.submit(work, arg) future_result.add_done_callback(when_done)
```

The user-supplied callback function receives an instance of Future that must be used to obtain the actual result (i.e., by calling its result() method). Although process pools can be easy to use, there are a number of important considerations to be made in designing larger programs. In no particular order:

- This technique for parallelization only works well for problems that can be trivially

decomposed into independent parts.

- Work must be submitted in the form of simple functions. Parallel execution of

instance methods, closures, or other kinds of constructs are not supported.

- Function arguments and return values must be compatible with pickle. Work is

carried out in a separate interpreter using interprocess communication. Thus, data exchanged between interpreters has to be serialized.

- Functions submitted for work should not maintain persistent state or have side

effects. With the exception of simple things such as logging, you don't really have any control over the behavior of child processes once started. Thus, to preserve your sanity, it is probably best to keep things simple and carry out work in pure-functions that don't alter their environment.

- Process pools are created by calling the fork() system call on Unix. This makes a

clone of the Python interpreter, including all of the program state at the time of the fork. On Windows, an independent copy of the interpreter that does not clone state is launched. The actual forking process does not occur until the first `pool.map()` or `pool.submit()` method is called.

- Great care should be made when combining process pools and programs that use

threads. In particular, you should probably create and launch process pools prior to the creation of any threads (e.g., create the pool in the main thread at program startup).

12.9 Python的全局锁问题

问题

You've heard about the Global Interpreter Lock (GIL), and are worried that it might be affecting the performance of your multithreaded program.

解决方案

Although Python fully supports thread programming, parts of the C implementation of the interpreter are not entirely thread safe to a level of allowing fully concurrent execution. In fact, the interpreter is protected by a so-called Global Interpreter Lock (GIL) that only allows one Python thread to execute at any given time. The most noticeable effect of the GIL is that multithreaded Python programs are not able to fully take advantage of multiple CPU cores (e.g., a computationally intensive application using more than one thread only runs on a single CPU).

Before discussing common GIL workarounds, it is important to emphasize that the GIL tends to only affect programs that are heavily CPU bound (i.e., dominated by computation). If your program is mostly doing I/O, such as network communication, threads are often a sensible choice because they're mostly going to spend their time sitting around waiting. In fact, you can create thousands of Python threads with barely a concern. Modern operating systems have no trouble running with that many threads, so it's simply not something you should worry much about. For CPU-bound programs, you really need to study the nature of the computation being performed. For instance, careful choice of the underlying algorithm may produce a far greater speedup than trying to parallelize an unoptimal algorithm with threads. Similarly, given that Python is interpreted, you might get a far greater speedup simply by moving performance-critical code into a C extension module. Extensions such as NumPy are also highly effective at speeding up certain kinds of calculations involving array data. Last, but not least, you might investigate alternative implementations, such as PyPy, which features optimizations such as a JIT compiler

(although, as of this writing, it does not yet support Python 3). It's also worth noting that threads are not necessarily used exclusively for performance. A CPU-bound program might be using threads to manage a graphical user interface, a network connection, or provide some other kind of service. In this case, the GIL can actually present more of a problem, since code that holds it for an excessively long period will cause annoying stalls in the non-CPU-bound threads. In fact, a poorly written C extension can actually make this problem worse, even though the computation part of the code might run faster than before. Having said all of this, there are two common strategies for working around the limitations of the GIL. First, if you are working entirely in Python, you can use the multiprocessing module to create a process pool and use it like a co-processor. For example, suppose you have the following thread code:

```
# Performs a large calculation (CPU bound) def some_work(args):
```

```
    ... return result
```

```
# A thread that calls the above function def some_thread():
```

```
    while True:
```

```
        ... r = some_work(args) ...
```

Here's how you would modify the code to use a pool:

```
# Processing pool (see below for initialization) pool = None
```

```
# Performs a large calculation (CPU bound) def some_work(args):
```

```
    ... return result
```

```
# A thread that calls the above function def some_thread():
```

```
    while True:
```

```
        ... r = pool.apply(some_work, (args)) ...
```

```
# Initialize the pool if __name__ == '__main__':
```

```
    import multiprocessing pool = multiprocessing.Pool()
```

This example with a pool works around the GIL using a neat trick. Whenever a thread wants to perform CPU-intensive work, it hands the work to the pool. The pool, in turn, hands the work to a separate Python interpreter running in a different process. While the thread is waiting for the result, it releases the GIL. Moreover, because the calculation is being performed in a separate interpreter, it's no longer bound by the restrictions of the GIL. On a

multicore system, you'll find that this technique easily allows you to take advantage of all the CPUs. The second strategy for working around the GIL is to focus on C extension programming. The general idea is to move computationally intensive tasks to C, independent of Python, and have the C code release the GIL while it's working. This is done by inserting special macros into the C code like this:

```
#include "Python.h" ...
```

```
PyObject *pyfunc(PyObject *self, PyObject *args) {  
    ... Py_BEGIN_ALLOW_THREADS // Threaded C code ... Py_END_ALLOW_THREADS ...  
}
```

If you are using other tools to access C, such as the ctypes library or Cython, you may not need to do anything. For example, ctypes releases the GIL when calling into C by default.

讨论

Many programmers, when faced with thread performance problems, are quick to blame the GIL for all of their ills. However, doing so is shortsighted and naive. Just as a real-

world example, mysterious "stalls" in a multithreaded network program might be caused by something entirely different (e.g., a stalled DNS lookup) rather than anything related to the GIL. The bottom line is that you really need to study your code to know if the GIL is an issue or not. Again, realize that the GIL is mostly concerned with CPU-bound processing, not I/O. If you are going to use a process pool as a workaround, be aware that doing so involves data serialization and communication with a different Python interpreter. For this to work, the operation to be performed needs to be contained within a Python function defined by the def statement (i.e., no lambdas, closures, callable instances, etc.), and the function arguments and return value must be compatible with pickle. Also, the amount of work to be performed must be sufficiently large to make up for the extra communication overhead. Another subtle aspect of pools is that mixing threads and process pools together can be a good way to make your head explode. If you are going to use both of these features together, it is often best to create the process pool as a singleton at program startup, prior to the creation of any threads. Threads will then use the same process pool for all of their computationally intensive work. For C extensions, the most important feature is maintaining isolation from the Python interpreter process. That is, if you're going to offload work from Python to C, you need to make sure the C code operates independently of Python. This means using no Python data structures and making no calls to Python's C API. Another consideration is that you want to make sure your C extension does enough work to make it all worthwhile. That is, it's much better if the extension can perform millions of calculations as opposed to just a few small calculations. Needless to say, these solutions to

working around the GIL don't apply to all possible problems. For instance, certain kinds of applications don't work well if separated into multiple processes, nor may you want to code parts in C. For these kinds of applications, you may have to come up with your own solution (e.g., multiple processes accessing shared memory regions, multiple interpreters running in the same process, etc.). Alternatively, you might look at some other implementations of the interpreter, such as PyPy. See Recipes 15.7 and 15.10 for additional information on releasing the GIL in C extensions.

12.10 定义一个 **Actor** 任务

问题

You'd like to define tasks with behavior similar to "actors" in the so-called "actor model."

解决方案

The "actor model" is one of the oldest and most simple approaches to concurrency and distributed computing. In fact, its underlying simplicity is part of its appeal. In a nutshell, an actor is a concurrently executing task that simply acts upon messages sent to it. In response to these messages, it may decide to send further messages to other actors. Communication with actors is one way and asynchronous. Thus, the sender of a message does not know when a message actually gets delivered, nor does it receive a response or acknowledgment that the message has been processed. Actors are straightforward to define using a combination of a thread and a queue. For example:

```
from queue import Queue from threading import Thread, Event
```

```
# Sentinel used for shutdown class ActorExit(Exception):
```

```
    pass
```

```
class Actor:
```

```
    def __init__(self):
```

```
        self._mailbox = Queue()
```

```
    def send(self, msg):
```

```
        """ Send a message to the actor """ self._mailbox.put(msg)
```

```
    def recv(self):
```

```
        """ Receive an incoming message """ msg = self._mailbox.get() if msg is ActorExit:
```

```
            raise ActorExit()
```

```
        return msg
```

```

def close(self):
    """ Close the actor, thus shutting it down """ self.send(ActorExit)

def start(self):
    """ Start concurrent execution """ self._terminated = Event() t =
    Thread(target=self._bootstrap)

    t.daemon = True t.start()

def _bootstrap(self):
    try:
        self.run()

    except ActorExit:
        pass

    finally:
        self._terminated.set()

def join(self):
    self._terminated.wait()

def run(self):
    """ Run method to be implemented by the user """ while True:

        msg = self.recv()

```

```
# Sample ActorTask class PrintActor(Actor):
```

```

def run(self):
    while True:
        msg = self.recv() print('Got:', msg)

```

```
# Sample use p = PrintActor() p.start() p.send('Hello') p.send('World') p.close() p.join()
```

In this example, Actor instances are things that you simply send a message to using their `send()` method. Under the covers, this places the message on a queue and hands it off to an internal thread that runs to process the received messages. The `close()` method is programmed to shut down the actor by placing a special sentinel value (`ActorExit`) on the queue. Users define new actors by inheriting from `Actor` and re-defining the `run()` method to implement their custom processing. The usage of the `ActorExit` exception is such that user-defined code can be programmed to catch the termination request and handle it if appropriate (the exception is raised by the `get()` method and propagated). If you relax the requirement of concurrent and asynchronous message delivery, actor-like objects can also be minimally defined by generators. For example:

```

def print_actor():
    while True:

```

```

try:
    msg = yield # Get a message print('Got:', msg)

except GeneratorExit:
    print('Actor terminating')

```

```

# Sample use p = print_actor() next(p) # Advance to the yield (ready to receive)
p.send('Hello') p.send('World') p.close()

```

讨论

Part of the appeal of actors is their underlying simplicity. In practice, there is just one core operation, `send()`. Plus, the general concept of a “message” in actor-based systems is something that can be expanded in many different directions. For example, you could pass tagged messages in the form of tuples and have actors take different courses of action like this:

```

class TaggedActor(Actor):
    def run(self):
        while True:
            tag, *payload = self.recv() getattr(self, 'do_'+tag)(*payload)

# Methods corresponding to different message tags def do_A(self, x):

    print('Running A', x)

    def do_B(self, x, y):
        print('Running B', x, y)

# Example a = TaggedActor() a.start() a.send(('A', 1)) # Invokes do_A(1) a.send(('B', 2, 3)) #
Invokes do_B(2,3)

```

As another example, here is a variation of an actor that allows arbitrary functions to be executed in a worker and results to be communicated back using a special `Result` object:

```

from threading import Event class Result:

```

```
def __init__(self):
    self._evt = Event()
    self._result = None
```

```
def set_result(self, value):
    self._result = value

    self._evt.set()
```

```
def result(self):
    self._evt.wait()
    return self._result
```

```
class Worker(Actor):
    def submit(self, func, *args, **kwargs):
        r = Result()
        self.send((func, args, kwargs, r))
        return r

    def run(self):
        while True:
            func, args, kwargs, r = self.recv()
            r.set_result(func(*args, **kwargs))
```

```
# Example use
worker = Worker()
worker.start()
r = worker.submit(pow, 2, 3)
print(r.result())
```

Last, but not least, the concept of “sending” a task a message is something that can be scaled up into systems involving multiple processes or even large distributed systems. For example, the `send()` method of an actor-like object could be programmed to transmit data on a socket connection or deliver it via some kind of messaging infrastructure (e.g., AMQP, ZMQ, etc.).

12.11 实现消息发布/订阅模型

问题

You have a program based on communicating threads and want them to implement publish/subscribe messaging.

解决方案

To implement publish/subscribe messaging, you typically introduce a separate “exchange” or “gateway” object that acts as an intermediary for all messages. That is, instead of directly sending a message from one task to another, a message is sent to the exchange and it delivers it to one or more attached tasks. Here is one example of a very simple exchange implementation:

```
from collections import defaultdict
```

```

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

```

```

# Dictionary of all created exchanges _exchanges = defaultdict(Exchange)

```

```

# Return the Exchange instance associated with a given name def get_exchange(name):

```

```

    return _exchanges[name]

```

An exchange is really nothing more than an object that keeps a set of active subscribers and provides methods for attaching, detaching, and sending messages. Each exchange is identified by a name, and the `get_exchange()` function simply returns the Exchange instance associated with a given name. Here is a simple example that shows how to use an exchange:

```

# Example of a task. Any object with a send() method

```

```

class Task:
    ... def send(self, msg):
        ...

```

```

task_a = Task() task_b = Task()

```

```

# Example of getting an exchange exc = get_exchange('name')

```

```

# Examples of subscribing tasks to it exc.attach(task_a) exc.attach(task_b)

```

```

# Example of sending messages exc.send('msg1') exc.send('msg2')

```

```

# Example of unsubscribing exc.detach(task_a) exc.detach(task_b)

```

Although there are many different variations on this theme, the overall idea is the same. Messages will be delivered to an exchange and the exchange will deliver them to attached subscribers.

讨论

The concept of tasks or threads sending messages to one another (often via queues) is easy to implement and quite popular. However, the benefits of using a public/subscribe (pub/sub) model instead are often overlooked. First, the use of an exchange can simplify much of the plumbing involved in setting up communicating threads. Instead of trying to wire threads together across multiple program modules, you only worry about connecting them to a known exchange. In some sense, this is similar to how the logging library works. In practice, it can make it easier to decouple various tasks in the program. Second, the ability of the exchange to broadcast messages to multiple subscribers opens up new communication patterns. For example, you could implement systems with redundant tasks, broadcasting, or fan-out. You could also build debugging and diagnostic tools that attach themselves to exchanges as ordinary subscribers. For example, here is a simple diagnostic class that would display sent messages:

```
class DisplayMessages:  
    def __init__(self):  
        self.count = 0  
  
    def send(self, msg):  
        self.count += 1 print('msg[{}]: {}'.format(self.count, msg))
```

```
exc = get_exchange('name') d = DisplayMessages() exc.attach(d)
```

Last, but not least, a notable aspect of the implementation is that it works with a variety of task-like objects. For example, the receivers of a message could be actors (as described in Recipe 12.10), coroutines, network connections, or just about anything that implements a proper send() method. One potentially problematic aspect of an exchange concerns the proper attachment and detachment of subscribers. In order to properly manage resources, every subscriber that attaches must eventually detach. This leads to a programming model similar to this:

```
exc = get_exchange('name') exc.attach(some_task) try:
```

```
...
```

```
finally:  
    exc.detach(some_task)
```

In some sense, this is similar to the usage of files, locks, and similar objects. Experience has shown that it is quite easy to forget the final detach() step. To simplify this, you might consider the use of the context-management protocol. For example, adding a subscribe() method to the exchange like this:

```
from contextlib import contextmanager  
from collections import defaultdict
```

```

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    @contextmanager
    def subscribe(self, *tasks):

        for task in tasks:
            self.attach(task)

        try:
            yield

        finally:
            for task in tasks:
                self.detach(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges _exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):

    return _exchanges[name]

# Example of using the subscribe() method
exc = get_exchange('name')
exc.subscribe(task_a, task_b):

    ...exc.send('msg1') exc.send('msg2') ...

# task_a and task_b detached here

```

Finally, it should be noted that there are numerous possible extensions to the exchange idea. For example, exchanges could implement an entire collection of message channels

or apply pattern matching rules to exchange names. Exchanges can also be extended into distributed computing applications (e.g., routing messages to tasks on different machines, etc.).

12.12 使用生成器代替线程

问题

You want to implement concurrency using generators (coroutines) as an alternative to system threads. This is sometimes known as user-level threading or green threading.

解决方案

To implement your own concurrency using generators, you first need a fundamental insight concerning generator functions and the yield statement. Specifically, the fundamental behavior of yield is that it causes a generator to suspend its execution. By suspending execution, it is possible to write a scheduler that treats generators as a kind of “task” and alternates their execution using a kind of cooperative task switching. To illustrate this idea, consider the following two generator functions using a simple yield:

```
# Two simple generator functions def countdown(n):
```

```
    while n > 0:
```

```
        print('T-minus', n) yield n -= 1
```

```
    print('Blastoff!')
```

```
def countup(n):
```

```
    x = 0 while x < n:
```

```
        print('Counting up', x) yield x += 1
```

These functions probably look a bit funny using yield all by itself. However, consider the following code that implements a simple task scheduler:

```
from collections import deque
```

```
class TaskScheduler:
```

```
    def __init__(self):
```

```
        self._task_queue = deque()
```

```
    def new_task(self, task):
```

```
        """ Admit a newly started task to the scheduler
```

```
        """ self._task_queue.append(task)
```

```
    def run(self):
```

```
        """ Run until there are no more tasks """ while self._task_queue:
```

```

task = self._task_queue.popleft() try:

    # Run until the next yield statement next(task) self._task_queue.append(task)

except StopIteration:
    # Generator is no longer executing pass

```

```

# Example use sched = TaskScheduler() sched.new_task(countdown(10))
sched.new_task(countdown(5)) sched.new_task(countup(15)) sched.run()

```

In this code, the TaskScheduler class runs a collection of generators in a round-robin manner—each one running until they reach a yield statement. For the sample, the output will be as follows:

```

T-minus 10 T-minus 5 Counting up 0 T-minus 9 T-minus 4 Counting up 1 T-minus 8 T-
minus 3 Counting up 2 T-minus 7 T-minus 2 ...

```

At this point, you’ve essentially implemented the tiny core of an “operating system” if you will. Generator functions are the tasks and the yield statement is how tasks signal that they want to suspend. The scheduler simply cycles over the tasks until none are left executing. In practice, you probably wouldn’t use generators to implement concurrency for some - thing as simple as shown. Instead, you might use generators to replace the use of threads when implementing actors (see Recipe 12.10) or network servers.

The following code illustrates the use of generators to implement a thread-free version of actors:

```

from collections import deque

```

```

class ActorScheduler:
    def __init__(self):
        self._actors = { } # Mapping of names to actors self._msg_queue = deque() # Message
        queue

    def new_actor(self, name, actor):
        """ Admit a newly started actor to the scheduler and give it a name """
        self._msg_queue.append((actor,None)) self._actors[name] = actor

    def send(self, name, msg):
        """ Send a message to a named actor """ actor = self._actors.get(name) if actor:

            self._msg_queue.append((actor,msg))

    def run(self):
        """ Run as long as there are pending messages. """ while self._msg_queue:

```

```
actor, msg = self._msg_queue.popleft() try:
```

```
    actor.send(msg)
```

```
except StopIteration:
```

```
    pass
```

```
# Example use if __name__ == '__main__':
```

```
def printer():
```

```
    while True:
```

```
        msg = yield print('Got:', msg)
```

```
def counter(sched):
```

```
    while True:
```

```
        # Receive the current count n = yield if n == 0:
```

```
            break
```

```
        # Send to the printer task sched.send('printer', n) # Send the next count to the  
        counter task (recursive)
```

```
        sched.send('counter', n-1)
```

```
sched = ActorScheduler() # Create the initial actors sched.new_actor('printer', printer())
```

```
sched.new_actor('counter', counter(sched))
```

```
# Send an initial message to the counter to initiate sched.send('counter', 10000)
```

```
sched.run()
```

The execution of this code might take a bit of study, but the key is the queue of pending messages. Essentially, the scheduler runs as long as there are messages to deliver. A remarkable feature is that the counter generator sends messages to itself and ends up in a recursive cycle not bound by Python's recursion limit. Here is an advanced example showing the use of generators to implement a concurrent network application:

```
from collections import deque from select import select
```

```
# This class represents a generic yield event in the scheduler class YieldEvent:
```

```
def handle_yield(self, sched, task):
```

```
    pass
```

```
def handle_resume(self, sched, task):
```

```
    pass
```

Task Scheduler class Scheduler:

def __init__(self):

```
self._numtasks = 0 # Total num of tasks
self._ready = deque() # Tasks ready to run
self._read_waiting = {} # Tasks waiting to read
self._write_waiting = {} # Tasks waiting to write
```

Poll for I/O events and restart waiting tasks **def _iopoll(self):**

```
rset,wset,eset = select(self._read_waiting,
self._write_waiting,[])
```

for r in rset:

```
    evt, task = self._read_waiting.pop(r)
    evt.handle_resume(self, task)
```

for w in wset:

```
    evt, task = self._write_waiting.pop(w)
    evt.handle_resume(self, task)
```

def new(self,task):

```
    """ Add a newly started task to the scheduler """
```

```
    self._ready.append((task, None))
    self._numtasks += 1
```

def add_ready(self, task, msg=None):

```
    """ Append an already started task to the ready queue. msg is what to send into the
    task when it resumes. """
    self._ready.append((task, msg))
```

Add a task to the reading set **def _read_wait(self, fileno, evt, task):**

```
    self._read_waiting[fileno] = (evt, task)
```

Add a task to the write set **def _write_wait(self, fileno, evt, task):**

```
    self._write_waiting[fileno] = (evt, task)
```

def run(self):

```
    """ Run the task scheduler until there are no tasks """
    while self._numtasks:
```

```
if not self._ready:
```

```
    self._iopoll()
```

```
task, msg = self._ready.popleft() try:
```

```
    # Run the coroutine to the next yield r = task.send(msg) if isinstance(r, YieldEvent):
```

```
        r.handle_yield(self, task)
```

```
else:
```

```
    raise RuntimeError('unrecognized yield event')
```

```
except StopIteration:
```

```
    self._numtasks -= 1
```

```
# Example implementation of coroutine-based socket I/O class ReadSocket(YieldEvent):
```

```
def __init__(self, sock, nbytes):
```

```
    self.sock = sock self.nbytes = nbytes
```

```
def handle_yield(self, sched, task):
```

```
    sched._read_wait(self.sock.fileno(), self, task)
```

```
def handle_resume(self, sched, task):
```

```
    data = self.sock.recv(self.nbytes) sched.add_ready(task, data)
```

```
class WriteSocket(YieldEvent):
```

```
    def __init__(self, sock, data):
```

```
        self.sock = sock self.data = data
```

```
    def handle_yield(self, sched, task):
```

```
        sched._write_wait(self.sock.fileno(), self, task)
```

```
    def handle_resume(self, sched, task):
```

```
        nsent = self.sock.send(self.data) sched.add_ready(task, nsent)
```

```
class AcceptSocket(YieldEvent):
```

```
    def __init__(self, sock):
```

```
        self.sock = sock
```

```
    def handle_yield(self, sched, task):
```

```
        sched._read_wait(self.sock.fileno(), self, task)
```

```
    def handle_resume(self, sched, task):
```

```
        r = self.sock.accept() sched.add_ready(task, r)
```

```
# Wrapper around a socket object for use with yield class Socket(object):
```

```
def __init__(self, sock):
```

```
    self._sock = sock
```

```
def recv(self, maxbytes):
```

```
    return ReadSocket(self._sock, maxbytes)
```

```
def send(self, data):
```

```
    return WriteSocket(self._sock, data)
```

```
def accept(self):
```

```
    return AcceptSocket(self._sock)
```

```
def __getattr__(self, name):
```

```
    return getattr(self._sock, name)
```

```
if __name__ == '__main__':
```

```
    from socket import socket, AF_INET, SOCK_STREAM import time
```

```
    # Example of a function involving generators. This should # be called using line = yield
```

```
    from readline(sock) def readline(sock):
```

```
        chars = [] while True:
```

```
            c = yield sock.recv(1) if not c:
```

```
                break
```

```
            chars.append(c) if c == b'n':
```

```
                break
```

```
        return b''.join(chars)
```

```
    # Echo server using generators class EchoServer:
```



```

def __init__(self,addr,sched):
    self.sched = sched sched.new(self.server_loop(addr))

def server_loop(self,addr):
    s = Socket(socket(AF_INET,SOCK_STREAM))

    s.bind(addr) s.listen(5) while True:

        c,a = yield s.accept() print('Got connection from ', a)
        self.sched.new(self.client_handler(Socket(c)))

def client_handler(self,client):
    while True:
        line = yield from readline(client) if not line:
            break

        line = b'GOT:' + line while line:
            nsent = yield client.send(line) line = line[nsent:]

    client.close() print('Client closed')

sched = Scheduler() EchoServer(('',16000),sched) sched.run()

```

This code will undoubtedly require a certain amount of careful study. However, it is essentially implementing a small operating system. There is a queue of tasks ready to run and there are waiting areas for tasks sleeping for I/O. Much of the scheduler involves moving tasks between the ready queue and the I/O waiting area.

讨论

When building generator-based concurrency frameworks, it is most common to work with the more general form of yield:

```

def some_generator():
    ... result = yield data ...

```

Functions that use yield in this manner are more generally referred to as “coroutines.” Within a scheduler, the yield statement gets handled in a loop as follows:

```

f = some_generator()

```

```

# Initial result. Is None to start since nothing has been computed result = None while True:

```

try:

```
data = f.send(result) result = ... do some calculation ...
```

except StopIteration:

```
break
```

The logic concerning the result is a bit convoluted. However, the value passed to `send()` defines what gets returned when the yield statement wakes back up. So, if a yield is going to return a result in response to data that was previously yielded, it gets returned on the next `send()` operation. If a generator function has just started, sending in a value of `None` simply makes it advance to the first yield statement. In addition to sending in values, it is also possible to execute a `close()` method on a generator. This causes a silent `GeneratorExit` exception to be raised at the yield statement, which stops execution. If desired, a generator can catch this exception and perform cleanup actions. It's also possible to use the `throw()` method of a generator to raise an arbitrary exception at the yield statement. A task scheduler might use this to communicate errors into running generators. The `yield from` statement used in the last example is used to implement coroutines that serve as subroutines or procedures to be called from other generators. Essentially, control transparently transfers to the new function. Unlike normal generators, a function that is called using `yield from` can return a value that becomes the result of the `yield from` statement. More information about `yield from` can be found in PEP 380. Finally, if programming with generators, it is important to stress that there are some major limitations. In particular, you get none of the benefits that threads provide. For instance, if you execute any code that is CPU bound or which blocks for I/O, it will suspend the entire task scheduler until the completion of that operation. To work around this, your only real option is to delegate the operation to a separate thread or process where it can run independently. Another limitation is that most Python libraries have not been written to work well with generator-based threading. If you take this approach, you may find that you need to write replacements for many standard library functions. As basic background on coroutines and the techniques utilized in this recipe, see PEP 342 and "A Curious Course on Coroutines and Concurrency". PEP 3156 also has a modern take on asynchronous I/O involving coroutines. In practice, it is extremely unlikely that you will write a low-level coroutine scheduler yourself. However, ideas surrounding coroutines are the basis for many popular libraries, including `gevent`, `greenlet`, `Stackless Python`, and similar projects.

12.13 多个线程队列轮询

问题

You have a collection of thread queues, and you would like to be able to poll them for incoming items, much in the same way as you might poll a collection of network connections for incoming data.

解决方案

A common solution to polling problems involves a little-known trick involving a hidden loopback network connection. Essentially, the idea is as follows: for each queue (or any object) that you want to poll, you create a pair of connected sockets. You then write on one of the sockets to signal the presence of data. The other socket is then passed to `select()` or a similar function to poll for the arrival of data. Here is some sample code that illustrates this idea:

```
import queue
import socket
import os
```

```
class PollableQueue(queue.Queue):
```

```
    def __init__(self):
```

```
        super().__init__() # Create a pair of connected sockets if os.name == 'posix':
```

```
            self._putsocket, self._getsocket = socket.socketpair()
```

```
        else:
```

```
            # Compatibility on non-POSIX systems
            server = socket.socket(socket.AF_INET,
                                   socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
```

```
            self._putsocket = socket.socket(socket.AF_INET,
                                             socket.SOCK_STREAM)
```

```
            self._putsocket.connect(server.getsockname())
            self._getsocket, _ = server.accept()
            server.close()
```

```
    def fileno(self):
```

```
        return self._getsocket.fileno()
```

```
    def put(self, item):
```

```
        super().put(item)
        self._putsocket.send(b'x')
```

```
    def get(self):
```

```
        self._getsocket.recv(1)
        return super().get()
```

In this code, a new kind of Queue instance is defined where there is an underlying pair of connected sockets. The `socketpair()` function on Unix machines can establish such sockets easily. On Windows, you have to fake it using code similar to that shown (it looks a bit weird, but a server socket is created and a client immediately connects to it afterward). The normal `get()` and `put()` methods are then redefined slightly to perform a small bit of I/O on these sockets. The `put()` method writes a single byte of data to one of the sockets after putting data on the queue. The `get()` method reads a single byte of data from the other socket when removing an item from the queue.

The `fileno()` method is what makes the queue pollable using a function such as `select()`. Essentially, it just exposes the underlying file descriptor of the socket used by the `get()` function. Here is an example of some code that defines a consumer which monitors multiple queues for incoming items:

```
import select import threading
```

```
def consumer(queues):
```

```
    """ Consumer that reads data on multiple queues simultaneously """ while True:
```

```
        can_read, _, _ = select.select(queues,[],[]) for r in can_read:
```

```
            item = r.get() print('Got:', item)
```

```
q1 = PollableQueue() q2 = PollableQueue() q3 = PollableQueue() t =
```

```
threading.Thread(target=consumer, args=(q1,q2,q3,)) t.daemon = True t.start()
```

```
# Feed data to the queues q1.put(1) q2.put(10) q3.put('hello') q2.put(15) ...
```

If you try it, you'll find that the consumer indeed receives all of the put items, regardless of which queues they are placed in.

讨论

The problem of polling non-file-like objects, such as queues, is often a lot trickier than it looks. For instance, if you don't use the socket technique shown, your only option is to write code that cycles through the queues and uses a timer, like this:

```
import time def consumer(queues):
```

```
    while True:
```

```
        for q in queues:
```

```
            if not q.empty():
```

```
                item = q.get() print('Got:', item)
```

```
            # Sleep briefly to avoid 100% CPU time.sleep(0.01)
```

This might work for certain kinds of problems, but it's clumsy and introduces other weird performance problems. For example, if new data is added to a queue, it won't be detected for as long as 10 milliseconds (an eternity on a modern processor). You run into even further problems if the preceding polling is mixed with the polling of other objects, such as network sockets. For example, if you want to poll both sockets and queues at the same time, you might have to use code like this:

```
import select
```

```
def event_loop(sockets, queues):
```

```
    while True:
```

```
# polling with a timeout
can_read, _, _ = select.select(sockets, [], [], 0.01)
for r in can_read:
    handle_read(r)

for q in queues:
    if not q.empty():
        item = q.get()
        print('Got:', item)
```

The solution shown solves a lot of these problems by simply putting queues on equal status with sockets. A single `select()` call can be used to poll for activity on both. It is not necessary to use timeouts or other time-based hacks to periodically check. Moreover, if data gets added to a queue, the consumer will be notified almost instantaneously. Although there is a tiny amount of overhead associated with the underlying I/O, it often is worth it to have better response time and simplified coding.

12.14 在Unix系统上面启动守护进程

问题

You would like to write a program that runs as a proper daemon process on Unix or Unix-like systems.

解决方案

Creating a proper daemon process requires a precise sequence of system calls and careful attention to detail. The following code shows how to define a daemon process along with the ability to easily stop it once launched:

```
#!/usr/bin/env python3 # daemon.py

import os
import sys

import atexit
import signal

def daemonize(pidfile, *, stdin='/dev/null',
              stdout='/dev/null', stderr='/dev/null'):

    if os.path.exists(pidfile):
        raise RuntimeError('Already running')

    # First fork (detaches from parent) try:
```

```

if os.fork() > 0:
    raise SystemExit(0) # Parent exit

except OSError as e:
    raise RuntimeError('fork #1 failed.')

os.chdir('/') os.umask(0) os.setsid() # Second fork (relinquish session leadership) try:

    if os.fork() > 0:
        raise SystemExit(0)

    except OSError as e:
        raise RuntimeError('fork #2 failed.')

# Flush I/O buffers sys.stdout.flush() sys.stderr.flush()

# Replace file descriptors for stdin, stdout, and stderr with open(stdin, 'rb', 0) as f:

    os.dup2(f.fileno(), sys.stdin.fileno())

with open(stdout, 'ab', 0) as f:
    os.dup2(f.fileno(), sys.stdout.fileno())

with open(stderr, 'ab', 0) as f:
    os.dup2(f.fileno(), sys.stderr.fileno())

# Write the PID file with open(pidfile,'w') as f:

    print(os.getpid(),file=f)

# Arrange to have the PID file removed on exit/signal atexit.register(lambda:
os.remove(pidfile))

# Signal handler for termination (required) def sigterm_handler(signo, frame):

    raise SystemExit(1)

signal.signal(signal.SIGTERM, sigterm_handler)

def main():
    import time sys.stdout.write('Daemon started with pid {}'.format(os.getpid())) while
    True:

        sys.stdout.write('Daemon Alive! {}'.format(time.ctime())) time.sleep(10)

if __name__ == '__main__':
    PIDFILE = '/tmp/daemon.pid'

    if len(sys.argv) != 2:
        print('Usage: {} [start|stop]'.format(sys.argv[0]), file=sys.stderr) raise SystemExit(1)

```

```

if sys.argv[1] == 'start':
    try:
        daemonize(PIDFILE,
                  stdout='/tmp/daemon.log', stderr='/tmp/dameon.log')

    except RuntimeError as e:
        print(e, file=sys.stderr) raise SystemExit(1)

    main()

elif sys.argv[1] == 'stop':
    if os.path.exists(PIDFILE):
        with open(PIDFILE) as f:
            os.kill(int(f.read()), signal.SIGTERM)

    else:
        print('Not running', file=sys.stderr) raise SystemExit(1)

else:
    print('Unknown command {!r}'.format(sys.argv[1]), file=sys.stderr) raise
    SystemExit(1)

```

To launch the daemon, the user would use a command like this:

```

bash % daemon.py start bash % cat /tmp/daemon.pid 2882 bash % tail -f /tmp/daemon.log
Daemon started with pid 2882 Daemon Alive! Fri Oct 12 13:45:37 2012 Daemon Alive! Fri
Oct 12 13:45:47 2012 ...

```

Daemon processes run entirely in the background, so the command returns immediately. However, you can view its associated pid file and log, as just shown. To stop the daemon, use:

```

bash % daemon.py stop bash %

```

讨论

This recipe defines a function `daemonize()` that should be called at program startup to make the program run as a daemon. The signature to `daemonize()` is using keyword-only arguments to make the purpose of the optional arguments more clear when used. This forces the user to use a call such as this:

```

daemonize('daemon.pid',
          stdin='/dev/null', stdout='/tmp/daemon.log', stderr='/tmp/daemon.log')

```

As opposed to a more cryptic call such as: `# Illegal. Must use keyword arguments`
`daemonize('daemon.pid',`

`'/dev/null', '/tmp/daemon.log', '/tmp/daemon.log')`

The steps involved in creating a daemon are fairly cryptic, but the general idea is as follows. First, a daemon has to detach itself from its parent process. This is the purpose of the first `os.fork()` operation and immediate termination by the parent. After the child has been orphaned, the call to `os.setsid()` creates an entirely new process session and sets the child as the leader. This also sets the child as the leader of a new process group and makes sure there is no controlling terminal. If this all sounds a bit too magical, it has to do with getting the daemon to detach properly from the terminal and making sure that things like signals don't interfere with its operation. The calls to `os.chdir()` and `os.umask(0)` change the current working directory and reset the file mode mask. Changing the directory is usually a good idea so that the daemon is no longer working in the directory from which it was launched. The second call to `os.fork()` is by far the more mysterious operation here. This step makes the daemon process give up the ability to acquire a new controlling terminal and provides even more isolation (essentially, the daemon gives up its session leadership and thus no longer has the permission to open controlling terminals). Although you could probably omit this step, it's typically recommended. Once the daemon process has been properly detached, it performs steps to reinitialize the standard I/O streams to point at files specified by the user. This part is actually somewhat tricky. References to file objects associated with the standard I/O streams are found in multiple places in the interpreter (`sys.stdout`, `sys.__stdout__`, etc.). Simply closing `sys.stdout` and reassigning it is not likely to work correctly, because there's no way to know if it will fix all uses of `sys.stdout`. Instead, a separate file object is opened, and the `os.dup2()` call is used to have it replace the file descriptor currently being used

by `sys.stdout`. When this happens, the original file for `sys.stdout` will be closed and the new one takes its place. It must be emphasized that any file encoding or text handling already applied to the standard I/O streams will remain in place. A common practice with daemon processes is to write the process ID of the daemon in a file for later use by other programs. The last part of the `daemonize()` function writes this file, but also arranges to have the file removed on program termination. The `atexit.register()` function registers a function to execute when the Python interpreter terminates. The definition of a signal handler for `SIGTERM` is also required for a graceful termination. The signal handler merely raises `SystemExit()` and nothing more. This might look unnecessary, but without it, termination signals kill the interpreter without performing the cleanup actions registered with `atexit.register()`. An example of code that kills the daemon can be found in the handling of the stop command at the end of the program. More information about writing daemon processes can be found in *Advanced Programming in the UNIX Environment, 2nd*

Edition, by W. Richard Stevens and Stephen A. Rago (Addison-Wesley, 2005). Although focused on C programming, all of the material is easily adapted to Python, since all of the required POSIX functions are available in the standard library.

第十三章：脚本编程与系统管理

许多人使用Python作为一个shell脚本的替代，用来实现常用系统任务的自动化，如文件的操作，系统的配置等。本章的主要目标是描述编写脚本时候经常遇到的一些功能。例如，解析命令行选项、获取有用的系统配置数据等等。第5章也包含了与文件和目录相关的一般信息。

Contents:

13.1 通过重定向/管道/文件接受输入

问题

You want a script you've written to be able to accept input using whatever mechanism is easiest for the user. This should include piping output from a command to the script, redirecting a file into the script, or just passing a filename, or list of filenames, to the script on the command line.

解决方案

Python's built-in fileinput module makes this very simple and concise. If you have a script that looks like this: `#!/usr/bin/env python3 import fileinput`

```
with fileinput.input() as f_input:
    for line in f_input:
        print(line, end='')
```

Then you can already accept input to the script in all of the previously mentioned ways. If you save this script as `filein.py` and make it executable, you can do all of the following and get the expected output:

```
$ ls | ./filein.py # Prints a directory listing to stdout.
$ ./filein.py /etc/passwd # Reads /etc/passwd to stdout.
$ ./filein.py < /etc/passwd # Reads /etc/passwd to stdout.
```

讨论

The `fileinput.input()` function creates and returns an instance of the `FileInput` class. In addition to containing a few handy helper methods, the instance can also be used as a context manager. So, to put all of this together, if we wrote a script that expected to be printing output from several files at once, we might have it include the filename and line number in the output, like this:

```
>>> import fileinput
>>> with fileinput.input('/etc/passwd') as f:
>>>     for line in f:
...         print(f.filename(), f.lineno(), line, end='')
...
/etc/passwd 1 ##
/etc/passwd 2 # User Database
/etc/passwd 3 #
```

<other output omitted>

Using it as a context manager ensures that the file is closed when it's no longer being used, and we leveraged a few handy `FileInput` helper methods here to get some extra information in the output.

13.2 终止程序并给出错误信息

问题

You want your program to terminate by printing a message to standard error and returning a nonzero status code.

解决方案

To have a program terminate in this manner, raise a `SystemExit` exception, but supply the error message as an argument. For example:

```
raise SystemExit('It failed!')
```

This will cause the supplied message to be printed to `sys.stderr` and the program to exit with a status code of 1.

讨论

This is a small recipe, but it solves a common problem that arises when writing scripts. Namely, to terminate a program, you might be inclined to write code like this:

```
import sys sys.stderr.write('It failed!\n') raise SystemExit(1)
```

None of the extra steps involving import or writing to `sys.stderr` are necessary if you simply supply the message to `SystemExit()` instead.

13.3 解析命令行选项

问题

You want to write a program that parses options supplied on the command line (found in `sys.argv`).

解决方案

The `argparse` module can be used to parse command-line options. A simple example will help to illustrate the essential features:

```
# search.py """ Hypothetical command-line tool for searching a collection of files for one or
more text patterns. """ import argparse parser =
argparse.ArgumentParser(description='Search some files')

parser.add_argument(dest='filenames', metavar='filename', nargs='*')

parser.add_argument('-p', '-pat', metavar='pattern', required=True,
    dest='patterns', action='append', help='text pattern to search for')

parser.add_argument('-v', dest='verbose', action='store_true',
    help='verbose mode')

parser.add_argument('-o', dest='outfile', action='store',
    help='output file')

parser.add_argument('-speed', dest='speed', action='store',
    choices={'slow', 'fast'}, default='slow', help='search speed')

args = parser.parse_args()

# Output the collected arguments print(args.filenames) print(args.patterns)
print(args.verbose) print(args.outfile) print(args.speed)
```

This program defines a command-line parser with the following usage:

```
bash % python3 search.py -h usage: search.py [-h] [-p pattern] [-v] [-o OUTFILE] [-speed {slow,fast}]
```

[filename [filename ...]]

Search some files

positional arguments:

filename

optional arguments:

-h, --help	show this help message and exit
-p pattern, --pat pattern	
	text pattern to search for
-v	verbose mode
-o OUTFILE	output file

-speed {slow,fast} search speed

The following session shows how data shows up in the program. Carefully observe the output of the print() statements.

```
bash % python3 search.py foo.txt bar.txt usage: search.py [-h] -p pattern [-v] [-o OUTFILE] [-speed {fast,slow}]
```

[filename [filename ...]]

search.py: error: the following arguments are required: -p/-pat

```
bash % python3 search.py -v -p spam -pat=eggs foo.txt bar.txt filenames = ['foo.txt', 'bar.txt'] patterns = ['spam', 'eggs'] verbose = True outfile = None speed = slow
```

```
bash % python3 search.py -v -p spam -pat=eggs foo.txt bar.txt -o results filenames = ['foo.txt', 'bar.txt'] patterns = ['spam', 'eggs'] verbose = True outfile = results speed = slow
```

```
bash % python3 search.py -v -p spam -pat=eggs foo.txt bar.txt -o results -speed=fast
```

```
filenames = ['foo.txt', 'bar.txt'] patterns = ['spam', 'eggs'] verbose = True outfile = results speed = fast
```

Further processing of the options is up to the program. Replace the print() functions with something more interesting.

讨论

The argparse module is one of the largest modules in the standard library, and has a huge number of configuration options. This recipe shows an essential subset that can be used and extended to get started. To parse options, you first create an ArgumentParser instance and add declarations for the options you want to support it using the add_argument() method. In each add_argument() call, the dest argument specifies the name of an attribute where the result of parsing will be placed. The metavar argument is used when generating help messages. The action argument specifies the processing associated with the argument and is often store for storing a value or append for collecting multiple argument values into a list. The following argument collects all of the extra command-line arguments into a list. It's being used to make a list of filenames in the example:

```
parser.add_argument(dest='filenames',metavar='filename', nargs='*')
```

The following argument sets a Boolean flag depending on whether or not the argument was provided:

```
parser.add_argument('-v', dest='verbose', action='store_true',  
                    help='verbose mode')
```

The following argument takes a single value and stores it as a string:

```
parser.add_argument('-o', dest='outfile', action='store',  
                    help='output file')
```

The following argument specification allows an argument to be repeated multiple times and all of the values append into a list. The required flag means that the argument must be supplied at least once. The use of -p and -pat mean that either argument name is acceptable.

```
parser.add_argument('-p', '-pat', metavar='pattern', required=True,  
                    dest='patterns', action='append', help='text pattern to search for')
```

Finally, the following argument specification takes a value, but checks it against a set of possible choices.

```
parser.add_argument('-speed', dest='speed', action='store',  
                    choices={'slow','fast'}, default='slow', help='search speed')
```

Once the options have been given, you simply execute the `parser.parse()` method. This will process the `sys.argv` value and return an instance with the results. The results

for each argument are placed into an attribute with the name given in the `dest` parameter to `add_argument()`. There are several other approaches for parsing command-line options. For example, you might be inclined to manually process `sys.argv` yourself or use the `getopt` module (which is modeled after a similarly named C library). However, if you take this approach, you'll simply end up replicating much of the code that `argparse` already provides. You may also encounter code that uses the `optparse` library to parse options. Although `optparse` is very similar to `argparse`, the latter is more modern and should be preferred in new projects.

13.4 运行时弹出密码输入提示

问题

You've written a script that requires a password, but since the script is meant for interactive use, you'd like to prompt the user for a password rather than hardcode it into the script.

解决方案

Python's `getpass` module is precisely what you need in this situation. It will allow you to very easily prompt for a password without having the keyed-in password displayed on the user's terminal. Here's how it's done:

```
import getpass

user = getpass.getuser() passwd = getpass.getpass()

if svc_login(user, passwd): # You must write svc_login()
    print('Yay!')
else:
    print('Boo!')
```

In this code, the `svc_login()` function is code that you must write to further process the password entry. Obviously, the exact handling is application-specific.

讨论

Note in the preceding code that `getpass.getuser()` doesn't prompt the user for their username. Instead, it uses the current user's login name, according to the user's shell environment, or as a last resort, according to the local system's password database (on platforms that support the `pwd` module).

If you want to explicitly prompt the user for their username, which can be more reliable, use the built-in input function:

```
user = input('Enter your username: ')
```

It's also important to remember that some systems may not support the hiding of the typed password input to the `getpass()` method. In this case, Python does all it can to forewarn you of problems (i.e., it alerts you that passwords will be shown in cleartext) before moving on.

13.5 获取终端的大小

问题

你需要知道当前终端的大小以便正确的格式化输出。

解决方案

使用 `os.get_terminal_size()` 函数来做到这一点。

代码示例:

```
>>> import os
>>> sz = os.get_terminal_size()
>>> sz
os.terminal_size(columns=80, lines=24)
>>> sz.columns
80
>>> sz.lines
24
>>>
```

讨论

有太多方式来得知终端大小了，从读取环境变量到执行底层的 `ioctl()` 函数等等。不过，为什么要去研究这些复杂的办法而不是仅仅调用一个简单的函数呢？

13.6 执行外部命令并获取它的输出

问题

You want to execute an external command and collect its output as a Python string.

解决方案

Use the `subprocess.check_output()` function. For example:

```
import subprocess out_bytes = subprocess.check_output(['netstat','-a'])
```

This runs the specified command and returns its output as a byte string. If you need to interpret the resulting bytes as text, add a further decoding step. For example:

```
out_text = out_bytes.decode('utf-8')
```

If the executed command returns a nonzero exit code, an exception is raised. Here is an example of catching errors and getting the output created along with the exit code:

try:

```
out_bytes = subprocess.check_output(['cmd','arg1','arg2'])
```

except subprocess.CalledProcessError as e:

```
out_bytes = e.output # Output generated before error code = e.returncode # Return code
```

By default, `check_output()` only returns output written to standard output. If you want both standard output and error collected, use the `stderr` argument:

```
out_bytes = subprocess.check_output(['cmd','arg1','arg2'],  
    stderr=subprocess.STDOUT)
```

If you need to execute a command with a timeout, use the `timeout` argument:

try:

```
out_bytes = subprocess.check_output(['cmd','arg1','arg2'], timeout=5)
```

except subprocess.TimeoutExpired as e:

...

Normally, commands are executed without the assistance of an underlying shell (e.g., `sh`, `bash`, etc.). Instead, the list of strings supplied are given to a low-level system command, such as `os.execve()`. If you want the command to be interpreted by a shell, supply it using a

simple string and give the `shell=True` argument. This is sometimes useful if you're trying to get Python to execute a complicated shell command involving pipes, I/O redirection, and other features. For example:

```
out_bytes = subprocess.check_output('grep python | wc > out', shell=True)
```

Be aware that executing commands under the shell is a potential security risk if arguments are derived from user input. The `shlex.quote()` function can be used to properly quote arguments for inclusion in shell commands in this case.

讨论

The `check_output()` function is the easiest way to execute an external command and get its output. However, if you need to perform more advanced communication with a

subprocess, such as sending it input, you'll need to take a different approach. For that, use the `subprocess.Popen` class directly. For example:

```
import subprocess
```

```
# Some text to send text = b"hello world this is a test goodbye"
```

```
# Launch a command with pipes p = subprocess.Popen(['wc'],
```

```
    stdout = subprocess.PIPE, stdin = subprocess.PIPE)
```

```
# Send the data and get the output stdout, stderr = p.communicate(text)
```

```
# To interpret as text, decode out = stdout.decode('utf-8') err = stderr.decode('utf-8')
```

The `subprocess` module is not suitable for communicating with external commands that expect to interact with a proper TTY. For example, you can't use it to automate tasks that ask the user to enter a password (e.g., a `ssh` session). For that, you would need to turn to a third-party module, such as those based on the popular "expect" family of tools (e.g., `pexpect` or similar).

13.7 复制或者移动文件和目录

问题

You need to copy or move files and directories around, but you don't want to do it by calling out to shell commands.

解决方案

The `shutil` module has portable implementations of functions for copying files and directories. The usage is extremely straightforward. For example:

```
import shutil

# Copy src to dst. (cp src dst) shutil.copy(src, dst)

# Copy files, but preserve metadata (cp -p src dst) shutil.copy2(src, dst)

# Copy directory tree (cp -R src dst) shutil.copytree(src, dst)

# Move src to dst (mv src dst) shutil.move(src, dst)
```

The arguments to these functions are all strings supplying file or directory names. The underlying semantics try to emulate that of similar Unix commands, as shown in the comments. By default, symbolic links are followed by these commands. For example, if the source file is a symbolic link, then the destination file will be a copy of the file the link points to. If you want to copy the symbolic link instead, supply the `follow_symlinks` keyword argument like this:

```
shutil.copy2(src, dst, follow_symlinks=False)
```

If you want to preserve symbolic links in copied directories, do this:

```
shutil.copytree(src, dst, symlinks=True)
```

The `copytree()` optionally allows you to ignore certain files and directories during the copy process. To do this, you supply an ignore function that takes a directory name and filename listing as input, and returns a list of names to ignore as a result. For example:

```
def ignore_pyc_files(dirname, filenames):
    return [name in filenames if name.endswith('.pyc')]
```

```
shutil.copytree(src, dst, ignore=ignore_pyc_files)
```

Since ignoring filename patterns is common, a utility function `ignore_patterns()` has already been provided to do it. For example:

```
shutil.copytree(src, dst, ignore=shutil.ignore_patterns('~', '.pyc'))
```

讨论

Using `shutil` to copy files and directories is mostly straightforward. However, one caution concerning file metadata is that functions such as `copy2()` only make a best effort in preserving this data. Basic information, such as access times, creation times, and permissions, will always be preserved, but preservation of owners, ACLs, resource forks, and other extended file metadata may or may not work depending on the underlying operating system and the user's own access permissions. You probably wouldn't want to use a function like `shutil.copytree()` to perform system backups. When working with filenames, make sure you use the functions in `os.path` for the greatest portability (especially if working with both Unix and Windows). For example:

```
>>> filename = '/Users/guido/programs/spam.py'
>>> import os.path
>>> os.path.basename(filename)
'spam.py'
>>> os.path.dirname(filename)
'/Users/guido/programs'
>>> os.path.split(filename)
('/Users/guido/programs', 'spam.py')
>>> os.path.join('/new/dir', os.path.basename(filename))
'/new/dir/spam.py'
>>> os.path.expanduser('~/' + os.path.basename(filename))
'/Users/guido/programs/spam.py'
>>>
```

One tricky bit about copying directories with `copytree()` is the handling of errors. For example, in the process of copying, the function might encounter broken symbolic links, files that can't be accessed due to permission problems, and so on. To deal with this, all exceptions encountered are collected into a list and grouped into a single exception that gets raised at the end of the operation. Here is how you would handle it:

try:

```
shutil.copytree(src, dst)
```

except `shutil.Error` as `e`:

```
for src, dst, msg in e.args[0]:
```

```
    # src is source name # dst is destination name # msg is error message from exception
    print(dst, src, msg)
```

If you supply the `ignore_dangling_symlinks=True` keyword argument, then `copy tree()` will ignore dangling symlinks. The functions shown in this recipe are probably the most commonly used. However, `shutil` has many more operations related to copying data. The documentation is definitely worth a further look. See the Python documentation.

13.8 创建和解压缩文件

问题

You need to create or unpack archives in common formats (e.g., `.tar`, `.tgz`, or `.zip`).

解决方案

The `shutil` module has two functions—`make_archive()` and `unpack_archive()`—that do exactly what you want. For example:

```
>>> import shutil
>>> shutil.unpack_archive('Python-3.3.0.tgz')
```

```
>>> shutil.make_archive('py33', 'zip', 'Python-3.3.0')
'/Users/beazley/Downloads/py33.zip'
>>>
```

The second argument to `make_archive()` is the desired output format. To get a list of supported archive formats, use `get_archive_formats()`. For example:

```
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"), ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'), ('zip', 'ZIP file')]
>>>
```

讨论

Python has other library modules for dealing with the low-level details of various archive formats (e.g., `tarfile`, `zipfile`, `gzip`, `bz2`, etc.). However, if all you're trying to do is make or extract an archive, there's really no need to go so low level. You can just use these high-level

functions in `shutil` instead. The functions have a variety of additional options for logging, dryruns, file permissions, and so forth. Consult the `shutil` library documentation for further details.

13.9 通过文件名查找文件

问题

You need to write a script that involves finding files, like a file renaming script or a log archiver utility, but you'd rather not have to call shell utilities from within your Python script, or you want to provide specialized behavior not easily available by "shelling out."

解决方案

To search for files, use the `os.walk()` function, supplying it with the top-level directory. Here is an example of a function that finds a specific filename and prints out the full path of all matches:

```
#!/usr/bin/env python3.3 import os

def findfile(start, name):
    for relpath, dirs, files in os.walk(start):
        if name in files:
            full_path = os.path.join(start, relpath, name)
            print(os.path.normpath(os.path.abspath(full_path)))

if __name__ == '__main__':
    findfile(sys.argv[1], sys.argv[2])
```

Save this script as `findfile.py` and run it from the command line, feeding in the starting point and the name as positional arguments, like this:

```
bash % ./findfile.py . myfile.txt
```

讨论

The `os.walk()` method traverses the directory hierarchy for us, and for each directory it enters, it returns a 3-tuple, containing the relative path to the directory it's inspecting, a list containing all of the directory names in that directory, and a list of filenames in that directory. For each tuple, you simply check if the target filename is in the files list. If it is, `os.path.join()` is used to put together a path. To avoid the possibility of weird looking paths

like `././foo//bar`, two additional functions are used to fix the result. The first is `os.path.abspath()`, which takes a path that might be relative and forms the absolute path, and the second is `os.path.normpath()`, which will normalize the path, thereby resolving issues with double slashes, multiple references to the current directory, and so on. Although this script is pretty simple compared to the features of the `find` utility found on UNIX platforms, it has the benefit of being cross-platform. Furthermore, a lot of additional functionality can be added in a portable manner without much more work. To illustrate, here is a function that prints out all of the files that have a recent modification time:

```
#!/usr/bin/env python3.3

import os
import time

def modified_within(top, seconds):
    now = time.time()
    for path, dirs, files in os.walk(top):
        for name in files:
            fullpath = os.path.join(path, name)
            if os.path.exists(fullpath):
                mtime = os.path.getmtime(fullpath)
                if mtime > (now - seconds):
                    print(fullpath)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: {} dir seconds'.format(sys.argv[0]))
        raise SystemExit(1)

    modified_within(sys.argv[1], float(sys.argv[2]))
```

It wouldn't take long for you to build far more complex operations on top of this little function using various features of the `os`, `os.path`, `glob`, and similar modules. See Recipes 5.11 and 5.13 for related recipes.

13.10 读取配置文件

问题

You want to read configuration files written in the common `.ini` configuration file format.

解决方案

The `configparser` module can be used to read configuration files. For example, suppose you have this configuration file:

; config.ini ; Sample configuration file

```
[installation] library=%(prefix)s/lib include=%(prefix)s/include bin=%(prefix)s/bin  
prefix=/usr/local
```

```
# Setting related to debug configuration [debug] log_errors=true show_warnings=False
```

```
[server] port: 8080 nworkers: 32 pid-file=/tmp/spam.pid root=/www/root signature:
```

Here is an example of how to read it and extract values:

```
>>> from configparser import ConfigParser  
>>> cfg = ConfigParser()  
>>> cfg.read('config.ini')  
['config.ini']  
>>> cfg.sections()  
['installation', 'debug', 'server']  
>>> cfg.get('installation', 'library')  
'/usr/local/lib'  
>>> cfg.getboolean('debug', 'log_errors')
```

```
True >>> cfg.getint('server', 'port') 8080 >>> cfg.getint('server', 'nworkers') 32 >>>  
print(cfg.get('server', 'signature'))
```

```
=====  
Brought to you by the Python Cookbook  
===== >>>
```

If desired, you can also modify the configuration and write it back to a file using the `cfg.write()` method. For example:

```
>>> cfg.set('server', 'port', '9000')  
>>> cfg.set('debug', 'log_errors', 'False')  
>>> import sys  
>>> cfg.write(sys.stdout)  
[installation]  
library = %(prefix)s/lib  
include = %(prefix)s/include  
bin = %(prefix)s/bin  
prefix = /usr/local
```

```
[debug] log_errors = False show_warnings = False
```

```
[server] port = 9000 nworkers = 32 pid-file = /tmp/spam.pid root = /www/root signature =
```

```
>>>
```

讨论

Configuration files are well suited as a human-readable format for specifying configuration data to your program. Within each config file, values are grouped into different sections (e.g., “installation,” “debug,” and “server,” in the example). Each section then specifies values for various variables in that section. There are several notable differences between a config file and using a Python source file for the same purpose. First, the syntax is much more permissive and “sloppy.” For example, both of these assignments are equivalent:

```
prefix=/usr/local prefix: /usr/local
```

The names used in a config file are also assumed to be case-insensitive. For example:

```
>>> cfg.get('installation', 'PREFIX')
'/usr/local'
>>> cfg.get('installation', 'prefix')
'/usr/local'
>>>
```

When parsing values, methods such as `getboolean()` look for any reasonable value. For example, these are all equivalent:

```
log_errors = true log_errors = TRUE log_errors = Yes log_errors = 1
```

Perhaps the most significant difference between a config file and Python code is that, unlike scripts, configuration files are not executed in a top-down manner. Instead, the file is read in its entirety. If variable substitutions are made, they are done after the fact. For example, in this part of the config file, it doesn’t matter that the `prefix` variable is assigned after other variables that happen to use it:

```
[installation] library=%(prefix)s/lib include=%(prefix)s/include bin=%(prefix)s/bin
prefix=/usr/local
```

An easily overlooked feature of `ConfigParser` is that it can read multiple configuration files together and merge their results into a single configuration. For example, suppose a user made their own configuration file that looked like this:


```
; ~/.config.ini [installation] prefix=/Users/beazley/test
```

```
[debug] log_errors=False
```

This file can be merged with the previous configuration by reading it separately. For example:

```
>>> # Previously read configuration
>>> cfg.get('installation', 'prefix')
'/usr/local'
```

```
>>> # Merge in user-specific configuration
>>> import os
>>> cfg.read(os.path.expanduser('~/.config.ini'))
['/Users/beazley/.config.ini']
```

```
>>> cfg.get('installation', 'prefix')
'/Users/beazley/test'
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.getboolean('debug', 'log_errors')
False
>>>
```

Observe how the override of the prefix variable affects other related variables, such as the setting of library. This works because variable interpolation is performed as late as possible. You can see this by trying the following experiment:

```
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.set('installation', 'prefix', '/tmp/dir')
>>> cfg.get('installation', 'library')
'/tmp/dir/lib'
>>>
```

Finally, it's important to note that Python does not support the full range of features you might find in an .ini file used by other programs (e.g., applications on Windows). Make sure you consult the configparser documentation for the finer details of the syntax and supported features.

13.11 给简单脚本增加日志功能

问题

You want scripts and simple programs to write diagnostic information to log files.

解决方案

The easiest way to add logging to simple programs is to use the logging module. For example:

```
import logging
```

```
def main():
```

```
    # Configure the logging system logging.basicConfig(
```

```
        filename='app.log', level=logging.ERROR
```

```
    )
```

```
    # Variables (to make the calls that follow work) hostname = 'www.python.org' item =
```

```
    'spam' filename = 'data.csv' mode = 'r'
```

```
    # Example logging calls (insert into your program) logging.critical('Host %s unknown',
```

```
    hostname) logging.error("Couldn't find %r", item) logging.warning('Feature is
```

```
    deprecated') logging.info('Opening file %r, mode=%r', filename, mode)
```

```
    logging.debug('Got here')
```

```
if __name__ == '__main__':
```

```
    main()
```

The five logging calls (`critical()`, `error()`, `warning()`, `info()`, `debug()`) represent different severity levels in decreasing order. The level argument to `basicConfig()` is a filter. All messages issued at a level lower than this setting will be ignored. The argument to each logging operation is a message string followed by zero or more arguments. When making the final log message, the `%` operator is used to format the message string using the supplied arguments. If you run this program, the contents of the file `app.log` will be as follows:

```
CRITICAL:root:Host www.python.org unknown ERROR:root:Could not find 'spam'
```

If you want to change the output or level of output, you can change the parameters to the `basicConfig()` call. For example:

```
logging.basicConfig(
```

```
filename='app.log', level=logging.WARNING, format='%(levelname)s:%(asctime)s:%(message)s')
```

As a result, the output changes to the following:

```
CRITICAL:2012-11-20 12:27:13,595:Host www.python.org unknown ERROR:2012-11-20 12:27:13,595:Could not find 'spam' WARNING:2012-11-20 12:27:13,595:Feature is deprecated
```

As shown, the logging configuration is hardcoded directly into the program. If you want to configure it from a configuration file, change the `basicConfig()` call to the following:

```
import logging import logging.config
```

```
def main():
```

```
    # Configure the logging system logging.config.fileConfig('logconfig.ini') ...
```

Now make a configuration file `logconfig.ini` that looks like this:

```
[loggers] keys=root
```

```
[handlers] keys=defaultHandler
```

```
[formatters] keys=defaultFormatter
```

```
[logger_root] level=INFO handlers=defaultHandler qualname=root
```

```
[handler_defaultHandler] class=FileHandler formatter=defaultFormatter args=('app.log', 'a')
```

```
[formatter_defaultFormatter] format='%(levelname)s:%(name)s:%(message)s'
```

If you want to make changes to the configuration, you can simply edit the `logconfig.ini` file as appropriate.

讨论

Ignoring for the moment that there are about a million advanced configuration options for the logging module, this solution is quite sufficient for simple programs and scripts. Simply make sure that you execute the `basicConfig()` call prior to making any logging calls, and your

program will generate logging output. If you want the logging messages to route to standard error instead of a file, don't supply any filename information to `basicConfig()`. For example, simply do this:

```
logging.basicConfig(level=logging.INFO)
```

One subtle aspect of `basicConfig()` is that it can only be called once in your program. If you later need to change the configuration of the logging module, you need to obtain the root logger and make changes to it directly. For example:

```
logging.getLogger().level = logging.DEBUG
```

It must be emphasized that this recipe only shows a basic use of the logging module. There are significantly more advanced customizations that can be made. An excellent resource for such customization is the “Logging Cookbook”.

13.12 给内库增加日志功能

问题

You would like to add a logging capability to a library, but don't want it to interfere with programs that don't use logging.

解决方案

For libraries that want to perform logging, you should create a dedicated logger object, and initially configure it as follows:

```
# somelib.py
```

```
import logging log = logging.getLogger(__name__) log.addHandler(logging.NullHandler())
```

```
# Example function (for testing) def func():
```

```
    log.critical('A Critical Error!') log.debug('A debug message')
```

With this configuration, no logging will occur by default. For example:

```
>>> import somelib
>>> somelib.func()
>>>
```

However, if the logging system gets configured, log messages will start to appear. For example:

```
>>> import logging
>>> logging.basicConfig()
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
>>>
```

讨论

Libraries present a special problem for logging, since information about the environment in which they are used isn't known. As a general rule, you should never write library code that tries to configure the logging system on its own or which makes assumptions about an already existing logging configuration. Thus, you need to take great care to provide isolation. The call to `getLogger(__name__)` creates a logger module that has the same name as the calling module. Since all modules are unique, this creates a dedicated logger that is likely to be separate from other loggers.

The `log.addHandler(logging.NullHandler())` operation attaches a null handler to the just created logger object. A null handler ignores all logging messages by default. Thus, if the library is used and logging is never configured, no messages or warnings will appear. One subtle feature of this recipe is that the logging of individual libraries can be independently configured, regardless of other logging settings. For example, consider the following code:

```
>>> import logging
>>> logging.basicConfig(level=logging.ERROR)
>>> import somelib
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
```

```
>>> # Change the logging level for 'somelib' only
>>> logging.getLogger('somelib').level=logging.DEBUG
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
DEBUG:somelib:A debug message
>>>
```

Here, the root logger has been configured to only output messages at the ERROR level or higher. However, the level of the logger for somelib has been separately configured to output debugging messages. That setting takes precedence over the global setting. The ability to change the logging settings for a single module like this can be a useful debugging tool, since you don't have to change any of the global logging settings—simply change the level for the one module where you want more output. The “Logging HOWTO” has more information about configuring the logging module and other useful tips.

13.13 记录程序执行的时间

问题

You want to be able to record the time it takes to perform various tasks.

解决方案

The time module contains various functions for performing timing-related functions. However, it's often useful to put a higher-level interface on them that mimics a stop watch. For example:

```
import time

class Timer:
    def __init__(self, func=time.perf_counter):
        self.elapsed = 0.0
        self._func = func
        self._start = None

    def start(self):
        if self._start is not None:
            raise RuntimeError('Already started')

        self._start = self._func()

    def stop(self):
        if self._start is None:
            raise RuntimeError('Not started')

        end = self._func()
        self.elapsed += end - self._start
        self._start = None

    def reset(self):
        self.elapsed = 0.0

    @property
    def running(self):
        return self._start is not None

    def __enter__(self):
```

```
self.start() return self
```

```
def __exit__(self, *args):  
    self.stop()
```

This class defines a timer that can be started, stopped, and reset as needed by the user. It keeps track of the total elapsed time in the `elapsed` attribute. Here is an example that shows how it can be used:

```
def countdown(n):  
    while n > 0:  
        n -= 1
```

```
# Use 1: Explicit start/stop t = Timer() t.start() countdown(1000000) t.stop()  
print(t.elapsed)
```

```
# Use 2: As a context manager with t:
```

```
    countdown(1000000)
```

```
print(t.elapsed)
```

```
with Timer() as t2:  
    countdown(1000000)
```

```
print(t2.elapsed)
```

讨论

This recipe provides a simple yet very useful class for making timing measurements and tracking elapsed time. It's also a nice illustration of how to support the context-management protocol and the `with` statement. One issue in making timing measurements concerns the underlying time function used to do it. As a general rule, the accuracy of timing measurements made with functions such as `time.time()` or `time.clock()` varies according to the operating system. In contrast, the `time.perf_counter()` function always uses the highest-resolution timer available on the system. As shown, the time recorded by the `Timer` class is made according to wall-clock time, and includes all time spent sleeping. If you only want the amount of CPU time used by the process, use `time.process_time()` instead. For example:

```
t = Timer(time.process_time) with t:
```

```
    countdown(1000000)
```

```
print(t.elapsed)
```

Both the `time.perf_counter()` and `time.process_time()` return a “time” in fractional seconds. However, the actual value of the time doesn’t have any particular meaning. To make sense of the results, you have to call the functions twice and compute a time difference. More examples of timing and profiling are given in Recipe 14.13.

13.14 限制内存和CPU的使用量

问题

You want to place some limits on the memory or CPU use of a program running on Unix system.

解决方案

The resource module can be used to perform both tasks. For example, to restrict CPU time, do the following:

```
import signal import resource import os

def time_exceeded(signo, frame):
    print("Time's up!") raise SystemExit(1)

def set_max_runtime(seconds):
    # Install the signal handler and set a resource limit soft, hard =
    resource.getrlimit(resource.RLIMIT_CPU) resource.setrlimit(resource.RLIMIT_CPU,
    (seconds, hard)) signal.signal(signal.SIGXCPU, time_exceeded)

if __name__ == '__main__':
    set_max_runtime(15) while True:

        pass
```

When this runs, the SIGXCPU signal is generated when the time expires. The program can then clean up and exit. To restrict memory use, put a limit on the total address space in use. For example:

```
import resource

def limit_memory(maxsize):
    soft, hard = resource.getrlimit(resource.RLIMIT_AS)
    resource.setrlimit(resource.RLIMIT_AS, (maxsize, hard))
```


With a memory limit in place, programs will start generating `MemoryError` exceptions when no more memory is available.

讨论

In this recipe, the `setrlimit()` function is used to set a soft and hard limit on a particular resource. The soft limit is a value upon which the operating system will typically restrict or notify the process via a signal. The hard limit represents an upper bound on the values that may be used for the soft limit. Typically, this is controlled by a system-wide parameter set by the system administrator. Although the hard limit can be lowered, it can never be raised by user processes (even if the process lowered itself). The `setrlimit()` function can additionally be used to set limits on things such as the number of child processes, number of open files, and similar system resources. Consult the documentation for the resource module for further details. Be aware that this recipe only works on Unix systems, and that it might not work on all of them. For example, when tested, it works on Linux but not on OS X.

13.15 启动一个WEB浏览器

问题

You want to launch a browser from a script and have it point to some URL that you specify.

解决方案

The `webbrowser` module can be used to launch a browser in a platform-independent manner. For example:

```
>>> import webbrowser
>>> webbrowser.open('http://www.python.org')
True
>>>
```

This opens the requested page using the default browser. If you want a bit more control over how the page gets opened, you can use one of the following functions:

```
>>> # Open the page in a new browser window
>>> webbrowser.open_new('http://www.python.org')
True
>>>
```

```
>>> # Open the page in a new browser tab
>>> webbrowser.open_new_tab('http://www.python.org')
True
>>>
```

These will try to open the page in a new browser window or tab, if possible and supported by the browser. If you want to open a page in a specific browser, you can use the `webbrowser.get()` function to specify a particular browser. For example:

```
>>> c = webbrowser.get('firefox')
>>> c.open('http://www.python.org')
True
>>> c.open_new_tab('http://docs.python.org')
True
>>>
```

A full list of supported browser names can be found in the Python documentation.

讨论

Being able to easily launch a browser can be a useful operation in many scripts. For example, maybe a script performs some kind of deployment to a server and you'd like to have it quickly launch a browser so you can verify that it's working. Or maybe a program writes data out in the form of HTML pages and you'd just like to fire up a browser to see the result. Either way, the `webbrowser` module is a simple solution.

第十四章：测试、调试和异常

试验还是很棒的，但是调试？就没那么有趣了。事实是，在Python测试代码之前没有编译器来分析你的代码，因此使的测试成为开发的一个重要部分。本章的目标是讨论一些关于测试、调试和异常处理的常见问题。但是并不是为测试驱动开发或者单元测试模块做一个简要的介绍。因此，笔者假定读者熟悉测试概念。

Contents:

14.1 测试输出到标准输出上

问题

You have a program that has a method whose output goes to standard Output (`sys.stdout`). This almost always means that it emits text to the screen. You'd like to write a test for your code to prove that, given the proper input, the proper output is displayed.

解决方案

Using the `unittest.mock` module's `patch()` function, it's pretty simple to mock out `sys.stdout` for just a single test, and put it back again, without messy temporary variables or leaking mocked-out state between test cases. Consider, as an example, the following function in a module `mymodule`:

```
# mymodule.py
```

```
def urlprint(protocol, host, domain):  
    url = '{}://{}.{}'.format(protocol, host, domain) print(url)
```

The built-in `print` function, by default, sends output to `sys.stdout`. In order to test that output is actually getting there, you can mock it out using a stand-in object, and then make assertions about what happened. Using the `unittest.mock` module's `patch()` method makes it convenient to replace objects only within the context of a running test, returning things to their original state immediately after the test is complete. Here's the test code for `mymodule`:

```
from io import StringIO from unittest import TestCase from unittest.mock import patch  
import mymodule
```

```
class TestURLPrint(TestCase):  
    def test_url_gets_to_stdout(self):  
        protocol = 'http' host = 'www' domain = 'example.com' expected_url = '{}://{}'.  
        {}n'.format(protocol, host, domain)  
  
        with patch('sys.stdout', new=StringIO()) as fake_out:  
            mymodule.urlprint(protocol, host, domain) self.assertEqual(fake_out.getvalue(),  
            expected_url)
```

讨论

The `urlprint()` function takes three arguments, and the test starts by setting up dummy arguments for each one. The `expected_url` variable is set to a string containing the expected output. To run the test, the `unittest.mock.patch()` function is used as a context manager to replace the value of `sys.stdout` with a `StringIO` object as a substitute. The `fake_out` variable

is the mock object that's created in this process. This can be used inside the body of the `with` statement to perform various checks. When the `with` statement completes, `patch` conveniently puts everything back the way it was before the test ever ran. It's worth noting that certain C extensions to Python may write directly to standard output, bypassing the setting of `sys.stdout`. This recipe won't help with that scenario, but it should work fine with pure Python code (if you need to capture I/O from such C extensions, you can do it by opening a temporary file and performing various tricks involving file descriptors to have standard output temporarily redirected to that file). More information about capturing IO in a string and `StringIO` objects can be found in Recipe 5.6.

14.2 在单元测试中给对象打补丁

问题

You're writing unit tests and need to apply patches to selected objects in order to make assertions about how they were used in the test (e.g., assertions about being called with certain parameters, access to selected attributes, etc.).

解决方案

The `unittest.mock.patch()` function can be used to help with this problem. It's a little unusual, but `patch()` can be used as a decorator, a context manager, or stand-alone. For example, here's an example of how it's used as a decorator:

```
from unittest.mock import patch import example
```

```
@patch('example.func') def test1(x, mock_func):
```

```
    example.func(x) # Uses patched example.func mock_func.assert_called_with(x)
```

It can also be used as a context manager:

```
with patch('example.func') as mock_func:
```

```
    example.func(x) # Uses patched example.func mock_func.assert_called_with(x)
```

Last, but not least, you can use it to patch things manually:

```
p = patch('example.func') mock_func = p.start() example.func(x)
mock_func.assert_called_with(x) p.stop()
```

If necessary, you can stack decorators and context managers to patch multiple objects. For example:

```
@patch('example.func1') @patch('example.func2') @patch('example.func3') def
test1(mock1, mock2, mock3):
```

...

```
def test2():
    with patch('example.patch1') as mock1,
        patch('example.patch2') as mock2, patch('example.patch3') as mock3:
```

...

讨论

patch() works by taking an existing object with the fully qualified name that you provide and replacing it with a new value. The original value is then restored after the completion of the decorated function or context manager. By default, values are replaced with MagicMock instances. For example:

```
>>> x = 42
>>> with patch('__main__.x'):
...     print(x)
...
<MagicMock name='x' id='4314230032'>
>>> x
42
>>>
```

However, you can actually replace the value with anything that you wish by supplying it as a second argument to patch():

```
>>> x
42
>>> with patch('__main__.x', 'patched_value'):
...     print(x)
...
patched_value
>>> x
42
>>>
```

The MagicMock instances that are normally used as replacement values are meant to mimic callables and instances. They record information about usage and allow you to make assertions. For example:

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock(return_value = 10)
>>> m(1, 2, debug=True)
10
>>> m.assert_called_with(1, 2, debug=True)
>>> m.assert_called_with(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "../unittest/mock.py", line 726, in assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: mock(1, 2)
Actual call: mock(1, 2, debug=True)
>>>
```

```
>>> m.upper.return_value = 'HELLO'
>>> m.upper('hello')
'HELLO'
>>> assert m.upper.called
```

```
>>> m.split.return_value = ['hello', 'world']
>>> m.split('hello world')
['hello', 'world']
>>> m.split.assert_called_with('hello world')
>>>
```

```
>>> m['blah']
<MagicMock name='mock.__getitem__()' id='4314412048'>
>>> m.__getitem__.called
True
>>> m.__getitem__.assert_called_with('blah')
>>>
```

Typically, these kinds of operations are carried out in a unit test. For example, suppose you have some function like this:

```
# example.py from urllib.request import urlopen import csv
```

```
def dowprices():
    u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=s11') lines =
```

```
(line.decode('utf-8') for line in u) rows = (row for row in csv.reader(lines) if len(row) == 2)
prices = { name:float(price) for name, price in rows } return prices
```

Normally, this function uses `urlopen()` to go fetch data off the Web and parse it. To unit test it, you might want to give it a more predictable dataset of your own creation, however. Here's an example using patching:

```
import unittest from unittest.mock import patch import io import example
```

```
sample_data = io.BytesIO(b'IBM,91.1r AA,13.25r MSFT,27.72r r')
```

```
class Tests(unittest.TestCase):
```

```
    @patch('example.urlopen', return_value=sample_data) def test_dowprices(self,
    mock_urlopen):
```

```
        p = example.dowprices() self.assertTrue(mock_urlopen.called) self.assertEqual(p,
```

```
            {'IBM': 91.1,
             'AA': 13.25, 'MSFT': 27.72})
```

```
if __name__ == '__main__':
    unittest.main()
```

In this example, the `urlopen()` function in the example module is replaced with a mock object that returns a `BytesIO()` containing sample data as a substitute. An important but subtle facet of this test is the patching of `example.urlopen` instead of `urllib.request.urlopen`. When you are making patches, you have to use the names as they are used in the code being tested. Since the example code uses `from urllib.request import urlopen`, the `urlopen()` function used by the `dowprices()` function is actually located in `example`. This recipe has really only given a very small taste of what's possible with the `unittest.mock` module. The official documentation is a must-read for more advanced features.

14.3 在单元测试中测试异常情况

问题

You want to write a unit test that cleanly tests if an exception is raised.

解决方案

To test for exceptions, use the `assertRaises()` method. For example, if you want to test that a function raised a `ValueError` exception, use this code:

```
import unittest

# A simple function to illustrate def parse_int(s):

    return int(s)
```

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaises(ValueError, parse_int, 'N/A')
```

If you need to test the exception's value in some way, then a different approach is needed. For example:

```
import errno

class TestIO(unittest.TestCase):
    def test_file_not_found(self):
        try:
            f = open('/file/not/found')

        except IOError as e:
            self.assertEqual(e.errno, errno.ENOENT)

        else:
            self.fail('IOError not raised')
```

讨论

The `assertRaises()` method provides a convenient way to test for the presence of an exception. A common pitfall is to write tests that manually try to do things with exceptions on their own. For instance:

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')

        except ValueError as e:
            self.assertEqual(type(e), ValueError)
```

The problem with such approaches is that it is easy to forget about corner cases, such as that when no exception is raised at all. To do that, you need to add an extra check for that situation, as shown here:

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
```



```

    r = parse_int('N/A')

    except ValueError as e:
        self.assertEqual(type(e), ValueError)

    else:
        self.fail('ValueError not raised')

```

The `assertRaises()` method simply takes care of these details, so you should prefer to use it. The one limitation of `assertRaises()` is that it doesn't provide a means for testing the value of the exception object that's created. To do that, you have to manually test it, as shown. Somewhere in between these two extremes, you might consider using the `assertRaisesRegex()` method, which allows you to test for an exception and perform a regular expression match against the exception's string representation at the same time. For example:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaisesRegex(ValueError, 'invalid literal .*',
                               parse_int, 'N/A')

```

A little-known fact about `assertRaises()` and `assertRaisesRegex()` is that they can also be used as context managers:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        with self.assertRaisesRegex(ValueError, 'invalid literal .*'):
            r = parse_int('N/A')

```

This form can be useful if your test involves multiple steps (e.g., setup) besides that of simply executing a callable.

14.4 将测试输出用日志记录到文件中

问题

You want the results of running unit tests written to a file instead of printed to standard output.

解决方案

A very common technique for running unit tests is to include a small code fragment like this at the bottom of your testing file:

```
import unittest
```

```
class MyTest(unittest.TestCase):
```

```
...
```

```
if __name__ == '__main__':  
    unittest.main()
```

This makes the test file executable, and prints the results of running tests to standard output. If you would like to redirect this output, you need to unwind the main() call a bit and write your own main() function like this:

```
import sys  
def main(out=sys.stderr, verbosity=2):  
  
    loader = unittest.TestLoader() suite =  
    loader.loadTestsFromModule(sys.modules[__name__])  
    unittest.TextTestRunner(out,verbosity=verbosity).run(suite)  
  
if __name__ == '__main__':  
    with open('testing.out', 'w') as f:  
        main(f)
```

讨论

The interesting thing about this recipe is not so much the task of getting test results redirected to a file, but the fact that doing so exposes some notable inner workings of the unittest module. At a basic level, the unittest module works by first assembling a test suite. This test suite consists of the different testing methods you defined. Once the suite has been assembled, the tests it contains are executed.

These two parts of unit testing are separate from each other. The unittest.TestLoader instance created in the solution is used to assemble a test suite. The loadTestsFromModule() is one of several methods it defines to gather tests. In this case, it scans a module for TestCase classes and extracts test methods from them. If you want something more fine-grained, the loadTestsFromTestCase() method (not shown) can be used to pull test methods from an individual class that inherits from TestCase. The TextTestRunner class is an example of a test runner class. The main purpose of this class is to execute the tests contained in a test suite. This class is the same test runner that sits behind the unittest.main() function. However, here we're giving it a bit of low-level configuration, including an output file and an elevated verbosity level. Although this recipe only consists of a few lines of code, it gives a hint as to how you might further customize the unittest framework. To customize how test suites are assembled, you would perform various operations using the TestLoader class. To customize how tests execute, you could make

custom test runner classes that emulate the functionality of TextTestRunner. Both topics are beyond the scope of what can be covered here. However, documentation for the unittest module has extensive coverage of the underlying protocols.

14.5 忽略或者期望测试失败

问题

You want to skip or mark selected tests as an anticipated failure in your unit tests.

解决方案

The unittest module has decorators that can be applied to selected test methods to control their handling. For example:

```
import unittest import os import platform
```

```
class Tests(unittest.TestCase):
```

```
    def test_0(self):
```

```
        self.assertTrue(True)
```

```
    @unittest.skip('skipped test') def test_1(self):
```

```
        self.fail('should have failed!')
```

```
    @unittest.skipIf(os.name=='posix', 'Not supported on Unix') def test_2(self):
```

```
        import winreg
```

```
    @unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific test') def test_3(self):
```

```
        self.assertTrue(True)
```

```
    @unittest.expectedFailure def test_4(self):
```

```
        self.assertEqual(2+2, 5)
```

```
if __name__ == '__main__':
```

```
    unittest.main()
```

If you run this code on a Mac, you'll get this output:

```
bash % python3 test_sample.py -v test_0 (__main__.Tests) ... ok test_1 (__main__.Tests) ...
skipped 'skipped test' test_2 (__main__.Tests) ... skipped 'Not supported on Unix' test_3
(__main__.Tests) ... ok test_4 (__main__.Tests) ... expected failure
```

Ran 5 tests in 0.002s

OK (skipped=2, expected failures=1)

讨论

The `skip()` decorator can be used to skip over a test that you don't want to run at all. `skipIf()` and `skipUnless()` can be a useful way to write tests that only apply to certain platforms or Python versions, or which have other dependencies. Use the `@expectedFailure` decorator to mark tests that are known failures, but for which you don't want the test framework to report more information. The decorators for skipping methods can also be applied to entire testing classes. For example:

```
@unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific tests') class
DarwinTests(unittest.TestCase):
```

...

14.6 处理多个异常

问题

You have a piece of code that can throw any of several different exceptions, and you need to account for all of the potential exceptions that could be raised without creating duplicate code or long, meandering code passages.

解决方案

If you can handle different exceptions all using a single block of code, they can be grouped together in a tuple like this:

```
try:
    client_obj.get_url(url)
except (URLError, ValueError, SocketTimeout):
    client_obj.remove_url(url)
```

In the preceding example, the `remove_url()` method will be called if any one of the listed exceptions occurs. If, on the other hand, you need to handle one of the exceptions differently, put it into its own `except` clause:

```
try:
    client_obj.get_url(url)
except (URLError, ValueError):
    client_obj.remove_url(url)
except SocketTimeout:
    client_obj.handle_url_timeout(url)
```

Many exceptions are grouped into an inheritance hierarchy. For such exceptions, you can catch all of them by simply specifying a base class. For example, instead of writing code like this:

```
try:
    f = open(filename)
except (FileNotFoundError, PermissionError):
    ...
```

you could rewrite the `except` statement as:

```
try:
    f = open(filename)
except OSError:
    ...
```

This works because `OSError` is a base class that's common to both the `FileNotFoundError` and `PermissionError` exceptions.

讨论

Although it's not specific to handling multiple exceptions per se, it's worth noting that you can get a handle to the thrown exception using the `as` keyword:

```
try:
    f = open(filename)
except OSError as e:
    if e.errno == errno.ENOENT:
        logger.error('File not found')
    elif e.errno == errno.EACCES:
        logger.error('Permission denied')
```

else:

```
logger.error('Unexpected error: %d', e.errno)
```

In this example, the `e` variable holds an instance of the raised `OSError`. This is useful if you need to inspect the exception further, such as processing it based on the value of an additional status code. Be aware that `except` clauses are checked in the order listed and that the first match executes. It may be a bit pathological, but you can easily create situations where multiple `except` clauses might match. For example:

```
>>> f = open('missing')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'missing'
>>> try:
...     f = open('missing')
... except OSError:
...     print('It failed')
... except FileNotFoundError:
...     print('File not found')
...
It failed
>>>
```

Here the `except FileNotFoundError` clause doesn't execute because the `OSError` is more general, matches the `FileNotFoundError` exception, and was listed first. As a debugging tip, if you're not entirely sure about the class hierarchy of a particular exception, you can quickly view it by inspecting the exception's `__mro__` attribute. For example:

```
>>> FileNotFoundError.__mro__
(<class 'FileNotFoundError'>, <class 'OSError'>, <class 'Exception'>,
 <class 'BaseException'>, <class 'object'>)
>>>
```

Any one of the listed classes up to `BaseException` can be used with the `except` statement.

14.7 捕获所有异常

问题

You want to write code that catches all exceptions.

解决方案

To catch all exceptions, write an exception handler for Exception, as shown here:

```
try:
```

```
...
```

```
except Exception as e:
```

```
... log('Reason:', e) # Important!
```

This will catch all exceptions save SystemExit, KeyboardInterrupt, and GeneratorExit. If you also want to catch those exceptions, change Exception to BaseException.

讨论

Catching all exceptions is sometimes used as a crutch by programmers who can't remember all of the possible exceptions that might occur in complicated operations. As such, it is also a very good way to write undebuggable code if you are not careful. Because of this, if you choose to catch all exceptions, it is absolutely critical to log or report the actual reason for the exception somewhere (e.g., log file, error message printed to screen, etc.). If you don't do this, your head will likely explode at some point. Consider this example:

```
def parse_int(s):
```

```
    try:
```

```
        n = int(v)
```

```
    except Exception:
```

```
        print("Couldn't parse")
```

If you try this function, it behaves like this:

```
>>> parse_int('n/a')
Couldn't parse
>>> parse_int('42')
Couldn't parse
>>>
```

At this point, you might be left scratching your head as to why it doesn't work. Now suppose the function had been written like this:

```
def parse_int(s):
```

```
    try:
```

```
        n = int(v)
```

```
    except Exception as e:
```

```
        print("Couldn't parse") print('Reason:', e)
```

In this case, you get the following output, which indicates that a programming mistake has been made:

```
>>> parse_int('42')
Couldn't parse
Reason: global name 'v' is not defined
>>>
```

All things being equal, it's probably better to be as precise as possible in your exception handling. However, if you must catch all exceptions, just make sure you give good diagnostic information or propagate the exception so that cause doesn't get lost.

14.8 创建自定义异常

问题

You're building an application and would like to wrap lower-level exceptions with custom ones that have more meaning in the context of your application.

解决方案

Creating new exceptions is easy—just define them as classes that inherit from Exception (or one of the other existing exception types if it makes more sense). For example, if you are writing code related to network programming, you might define some custom exceptions like this:

```
class NetworkError(Exception):
    pass

class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

class ProtocolError(NetworkError):
    pass
```

Users could then use these exceptions in the normal way. For example:

```
try:
    msg = s.recv()
except TimeoutError as e:
    ...
```



```
except ProtocolError as e:
```

```
...
```

讨论

Custom exception classes should almost always inherit from the built-in Exception class, or inherit from some locally defined base exception that itself inherits from Exception. Although all exceptions also derive from BaseException, you should not use this as a base class for new exceptions. BaseException is reserved for system-exiting exceptions, such as KeyboardInterrupt or SystemExit, and other exceptions that should signal the application to exit. Therefore, catching these exceptions is not the

intended use case. Assuming you follow this convention, it follows that inheriting from BaseException causes your custom exceptions to not be caught and to signal an imminent application shutdown! Having custom exceptions in your application and using them as shown makes your application code tell a more coherent story to whoever may need to read the code. One design consideration involves the grouping of custom exceptions via inheritance. In complicated applications, it may make sense to introduce further base classes that group different classes of exceptions together. This gives the user a choice of catching a narrowly specified error, such as this:

```
try:
```

```
    s.send(msg)
```

```
except ProtocolError:
```

```
...
```

It also gives the ability to catch a broad range of errors, such as the following:

```
try:
```

```
    s.send(msg)
```

```
except NetworkError:
```

```
...
```

If you are going to define a new exception that overrides the `__init__()` method of Exception, make sure you always call `Exception.__init__()` with all of the passed arguments. For example:

```
class CustomError(Exception):
```

```
    def __init__(self, message, status):
```

```
        super().__init__(message, status) self.message = message self.status = status
```

This might look a little weird, but the default behavior of Exception is to accept all arguments passed and to store them in the .args attribute as a tuple. Various other libraries and parts of Python expect all exceptions to have the .args attribute, so if you skip this step, you might find that your new exception doesn't behave quite right in certain contexts. To illustrate the use of .args, consider this interactive session with the built-in RuntimeError exception, and notice how any number of arguments can be used with the raise statement:

```
>>> try:
...     raise RuntimeError('It failed')
... except RuntimeError as e:
...     print(e.args)
...
('It failed',)
>>> try:
...     raise RuntimeError('It failed', 42, 'spam')
... except RuntimeError as e:
```

```
...print(e.args) ... ('It failed', 42, 'spam') >>>
```

For more information on creating your own exceptions, see the Python documentation.

14.9 捕获异常后抛出另外的异常

问题

You want to raise an exception in response to catching a different exception, but want to include information about both exceptions in the traceback.

解决方案

To chain exceptions, use the raise from statement instead of a simple raise statement. This will give you information about both errors. For example:

```
>>> def example():
...     try:
...         int('N/A')
...     except ValueError as e:
...         raise RuntimeError('A parsing error occurred') from e...
>>>
example()
Traceback (most recent call last):
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
```

The above exception was the direct cause of the following exception:

Traceback (most recent call last):

File "<stdin>", line 1, in <module> File "<stdin>", line 5, in example

RuntimeError: A parsing error occurred >>>

As you can see in the traceback, both exceptions are captured. To catch such an exception, you would use a normal except statement. However, you can look at the `__cause__` attribute of the exception object to follow the exception chain should you wish. For example: try:

```
example()
```

except RuntimeError as e:

```
print("It didn't work:", e)
```

if e.__cause__:

```
print('Cause:', e.__cause__)
```

An implicit form of chained exceptions occurs when another exception gets raised inside an except block. For example:

```
>>> def example2():
...     try:
...         int('N/A')
...     except ValueError as e:
...         print("Couldn't parse:", err)
...
>>>
>>> example2()
Traceback (most recent call last):
  File "<stdin>", line 3, in example2
ValueError: invalid literal for int() with base 10: 'N/A'
```

During handling of the above exception, another exception occurred:

Traceback (most recent call last):

File "<stdin>", line 1, in <module> File "<stdin>", line 5, in example2

NameError: global name 'err' is not defined >>>

In this example, you get information about both exceptions, but the interpretation is a bit different. In this case, the `NameError` exception is raised as the result of a programming error, not in direct response to the parsing error. For this case, the `__cause__` attribute of an exception is not set. Instead, a `__context__` attribute is set to the prior exception. If, for some reason, you want to suppress chaining, use `raise from None`:

```
>>> def example3():
...     try:
...         int('N/A')
...     except ValueError:
...         raise RuntimeError('A parsing error occurred') from None...
>>>
example3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example3
RuntimeError: A parsing error occurred
>>>
```

讨论

In designing code, you should give careful attention to use of the `raise` statement inside of other `except` blocks. In most cases, such `raise` statements should probably be changed to `raise from` statements. That is, you should prefer this style:

```
try:
```

```
...
```

```
except SomeException as e:
```

```
    raise DifferentException() from e
```

The reason for doing this is that you are explicitly chaining the causes together. That is, the `DifferentException` is being raised in direct response to getting a `SomeException`. This relationship will be explicitly stated in the resulting traceback. If you write your code in the following style, you still get a chained exception, but it's often not clear if the exception chain was intentional or the result of an unforeseen programming error:

```
try:
```

```
...
```

```
except SomeException:
```

```
    raise DifferentException()
```

When you use `raise from`, you're making it clear that you meant to raise the second exception. Resist the urge to suppress exception information, as shown in the last example. Although suppressing exception information can lead to smaller tracebacks, it also discards information that might be useful for debugging. All things being equal, it's often best to keep as much information as possible.

14.10 重新抛出最后的异常

问题

You caught an exception in an `except` block, but now you want to reraise it.

解决方案

Simply use the `raise` statement all by itself. For example:

```
>>> def example():
...     try:
...         int('N/A')
...     except ValueError:
...         print("Didn't work")
...         raise
... 
```

```
>>> example()
Didn't work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
>>> 
```

讨论

This problem typically arises when you need to take some kind of action in response to an exception (e.g., logging, cleanup, etc.), but afterward, you simply want to propagate the exception along. A very common use might be in catch-all exception handlers:

```
try:
```

```
...
```

except Exception as e:

```
# Process exception information in some way ...
```

```
# Propagate the exception raise
```

14.11 输出警告信息

问题

You want to have your program issue warning messages (e.g., about deprecated features or usage problems).

解决方案

To have your program issue a warning message, use the `warnings.warn()` function. For example:

```
import warnings
```

```
def func(x, y, logfile=None, debug=False):
```

```
    if logfile is not None:
```

```
        warnings.warn('logfile argument deprecated', DeprecationWarning)
```

```
    ...
```

The arguments to `warn()` are a warning message along with a warning class, which is typically one of the following: `UserWarning`, `DeprecationWarning`, `SyntaxWarning`, `RuntimeWarning`, `ResourceWarning`, or `FutureWarning`. The handling of warnings depends on how you have executed the interpreter and other configuration. For example, if you run Python with the `-W all` option, you'll get output such as the following:

```
bash % python3 -W all example.py example.py:5: DeprecationWarning: logfile  
argument is deprecated
```

```
warnings.warn('logfile argument is deprecated', DeprecationWarning)
```

Normally, warnings just produce output messages on standard error. If you want to turn warnings into exceptions, use the `-W error` option:

```
bash % python3 -W error example.py Traceback (most recent call last):
```

```
File "example.py", line 10, in <module>
```

```
func(2, 3, logfile='log.txt')
```

```
File "example.py", line 5, in func
```

```
warnings.warn('logfile argument is deprecated', DeprecationWarning)
```

```
DeprecationWarning: logfile argument is deprecated bash %
```

讨论

Issuing a warning message is often a useful technique for maintaining software and assisting users with issues that don't necessarily rise to the level of being a full-fledged exception. For example, if you're going to change the behavior of a library or framework, you can start issuing warning messages for the parts that you're going to change while still providing backward compatibility for a time. You can also warn users about problematic usage issues in their code. As another example of a warning in the built-in library, here is an example of a warning message generated by destroying a file without closing it:

```
>>> import warnings
>>> warnings.simplefilter('always')
>>> f = open('/etc/passwd')
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.TextIOWrapper name='/etc/passwd'
mode='r' encoding='UTF-8'>
>>>
```

By default, not all warning messages appear. The `-W` option to Python can control the output of warning messages. `-W all` will output all warning messages, `-W ignore` ignores all warnings, and `-W error` turns warnings into exceptions. As an alternative, you can use the `warnings.simplefilter()` function to control output, as just shown. An argument of `always` makes all warning messages appear, `ignore` ignores all warnings, and `error` turns warnings into exceptions. For simple cases, this is all you really need to issue warning messages. The `warnings` module provides a variety of more advanced configuration options related to the filtering and handling of warning messages. See the Python documentation for more information.

14.12 调试基本的程序崩溃错误

问题

Your program is broken and you'd like some simple strategies for debugging it.

解决方案

If your program is crashing with an exception, running your program as `python3 -i someprogram.py` can be a useful tool for simply looking around. The `-i` option starts an interactive shell as soon as a program terminates. From there, you can explore the environment. For example, suppose you have this code:

```
# sample.py

def func(n):
    return n + 10

func('Hello')
```

Running `python3 -i` produces the following:

```
bash % python3 -i sample.py Traceback (most recent call last):
```

```
File "sample.py", line 6, in <module>
    func('Hello')
```

```
File "sample.py", line 4, in func
    return n + 10
```

```
TypeError: Can't convert 'int' object to str implicitly >>> func(10) 20 >>>
```

If you don't see anything obvious, a further step is to launch the Python debugger after a crash. For example:

```
>>> import pdb
>>> pdb.pm()
> sample.py(4)func()
-> return n + 10
(Pdb) w
    sample.py(6)<module>()
-> func('Hello')
> sample.py(4)func()
-> return n + 10
(Pdb) print n
'Hello'
(Pdb) q
>>>
```


If your code is deeply buried in an environment where it is difficult to obtain an interactive shell (e.g., in a server), you can often catch errors and produce tracebacks yourself. For example:

```
import traceback import sys
```

try:

```
    func(arg)
```

except:

```
    print('*** AN ERROR OCCURRED ***') traceback.print_exc(file=sys.stderr)
```

If your program isn't crashing, but it's producing wrong answers or you're mystified by how it works, there is often nothing wrong with just injecting a few `print()` calls in places of interest. However, if you're going to do that, there are a few related techniques of interest. First, the `traceback.print_stack()` function will create a stack track of your program immediately at that point. For example:

```
>>> def sample(n):
...     if n > 0:
...         sample(n-1)
...     else:
...         traceback.print_stack(file=sys.stderr)
...
>>> sample(5)
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 5, in sample
>>>
```

Alternatively, you can also manually launch the debugger at any point in your program using `pdb.set_trace()` like this:

```
import pdb
```

def func(arg):

```
    ...pdb.set_trace() ...
```

This can be a useful technique for poking around in the internals of a large program and answering questions about the control flow or arguments to functions. For instance, once the debugger starts, you can inspect variables using `print` or type a command such as `w` to get the stack traceback.

讨论

Don't make debugging more complicated than it needs to be. Simple errors can often be resolved by merely knowing how to read program tracebacks (e.g., the actual error is usually the last line of the traceback). Inserting a few selected `print()` functions in your code can also work well if you're in the process of developing it and you simply want some diagnostics (just remember to remove the statements later). A common use of the debugger is to inspect variables inside a function that has crashed. Knowing how to enter the debugger after such a crash has occurred is a useful skill to know. Inserting statements such as `pdb.set_trace()` can be useful if you're trying to unravel an extremely complicated program where the underlying control flow isn't obvious. Essentially, the program will run until it hits the `set_trace()` call, at which point it will immediately enter the debugger. From there, you can try to make more sense of it. If you're using an IDE for Python development, the IDE will typically provide its own debugging interface on top of or in place of `pdb`. Consult the manual for your IDE for more information.

14.13 给你的程序做基准测试

问题

You would like to find out where your program spends its time and make timing measurements.

解决方案

If you simply want to time your whole program, it's usually easy enough to use something like the Unix `time` command. For example:

```
bash % time python3 someprogram.py real 0m13.937s user 0m12.162s sys 0m0.098s bash %
```

On the other extreme, if you want a detailed report showing what your program is doing, you can use the `cProfile` module:

```
bash % python3 -m cProfile someprogram.py
      859647 function calls in 16.016 CPU seconds
```

Ordered by: standard name

```
ncalls tottime percall cumtime percall filename:lineno(function) 263169 0.080 0.000
0.080 0.000 someprogram.py:16(frange)
```

```
513 0.001 0.000 0.002 0.000 someprogram.py:30(generate_mandel)
```

```
262656 0.194 0.000 15.295 0.000 someprogram.py:32(<genexpr>)
1 0.036 0.036 16.077 16.077 someprogram.py:4(<module>)
```

```
262144 15.021 0.000 15.021 0.000 someprogram.py:4(in_mandelbrot)
1 0.000 0.000 0.000 0.000 os.py:746(urandom) 1 0.000 0.000 0.000 0.000
png.py:1056(_readable) 1 0.000 0.000 0.000 0.000 png.py:1073(Reader) 1
0.227 0.227 0.438 0.438 png.py:163(<module>)
```

```
512 0.010 0.000 0.010 0.000 png.py:200(group)
```

```
...
```

```
bash %
```

More often than not, profiling your code lies somewhere in between these two extremes. For example, you may already know that your code spends most of its time in a few selected functions. For selected profiling of functions, a short decorator can be useful. For example:

```
# timethis.py
```

```
import time from functools import wraps
```

```
def timethis(func):
```

```
    @wraps(func) def wrapper(*args, **kwargs):
```

```
        start = time.perf_counter() r = func(*args, **kwargs) end = time.perf_counter()
        print('{}.{}: {}'.format(func.__module__, func.__name__, end - start)) return r
```

```
    return wrapper
```

To use this decorator, you simply place it in front of a function definition to get timings from it. For example:

```
>>> @timethis
... def countdown(n):
...     while n > 0:
...         n -= 1
...
>>> countdown(10000000)
__main__.countdown : 0.803001880645752
>>>
```

To time a block of statements, you can define a context manager. For example:

```
from contextlib import contextmanager
```

```
@contextmanager def timeblock(label):
```

```
    start = time.perf_counter() try:
```

```
        yield
```

```
    finally:
```

```
        end = time.perf_counter() print('{} : {}'.format(label, end - start))
```

Here is an example of how the context manager works:

```
>>> with timeblock('counting'):
...     n = 10000000
...     while n > 0:
...         n -= 1
...
counting : 1.5551159381866455
>>>
```

For studying the performance of small code fragments, the `timeit` module can be useful. For example:

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2)', 'import math')
0.1432319980012835
>>> timeit('sqrt(2)', 'from math import sqrt')
0.10836604500218527
>>>
```

`timeit` works by executing the statement specified in the first argument a million times and measuring the time. The second argument is a setup string that is executed to set up the environment prior to running the test. If you need to change the number of iterations, supply a number argument like this:

```
>>> timeit('math.sqrt(2)', 'import math', number=10000000)
1.434852126003534
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000000)
1.0270336690009572
>>>
```

讨论

When making performance measurements, be aware that any results you get are approximations. The `time.perf_counter()` function used in the solution provides the highest-resolution timer possible on a given platform. However, it still measures wall-clock time, and can be impacted by many different factors, such as machine load. If you are interested in process time as opposed to wall-clock time, use `time.process_time()` instead. For example:

```
from functools import wraps def timethis(func):
```

```
    @wraps(func) def wrapper(*args, **kwargs):
```

```
        start = time.process_time() r = func(*args, **kwargs) end = time.process_time()
        print('{}:{}'.format(func.__module__, func.__name__, end - start)) return r
```

```
    return wrapper
```

Last, but not least, if you're going to perform detailed timing analysis, make sure to read the documentation for the `time`, `timeit`, and other associated modules, so that you have an understanding of important platform-related differences and other pitfalls. See Recipe 13.13 for a related recipe on creating a stopwatch timer class.

14.14 让你的程序跑的更快

问题

Your program runs too slow and you'd like to speed it up without the assistance of more extreme solutions, such as C extensions or a just-in-time (JIT) compiler.

解决方案

While the first rule of optimization might be to “not do it,” the second rule is almost certainly “don’t optimize the unimportant.” To that end, if your program is running slow, you might start by profiling your code as discussed in Recipe 14.13. More often than not, you’ll find that your program spends its time in a few hotspots, such as inner data processing loops. Once you’ve identified those locations, you can use the no-nonsense techniques presented in the following sections to make your program run faster.

Use functions A lot of programmers start using Python as a language for writing simple scripts. When writing scripts, it is easy to fall into a practice of simply writing code with very little structure. For example:

```
# somescript.py

import sys import csv

with open(sys.argv[1]) as f:
    for row in csv.reader(f):

        # Some kind of processing ...
```

A little-known fact is that code defined in the global scope like this runs slower than code defined in a function. The speed difference has to do with the implementation of local versus global variables (operations involving locals are faster). So, if you want to make the program run faster, simply put the scripting statements in a function:

```
# somescript.py import sys import csv

def main(filename):
    with open(filename) as f:
        for row in csv.reader(f):
            # Some kind of processing ...

main(sys.argv[1])
```

The speed difference depends heavily on the processing being performed, but in our experience, speedups of 15-30% are not uncommon.

Selectively eliminate attribute access Every use of the dot (.) operator to access attributes comes with a cost. Under the covers, this triggers special methods, such as `__getattr__()` and `__getattribute__()`, which often lead to dictionary lookups. You can often avoid attribute lookups by using the `from module import name` form of import as well as making selected use of bound methods. To illustrate, consider the following code fragment:

```
import math
```

```

def compute_roots(nums):
    result = []
    for n in nums:
        result.append(math.sqrt(n))

    return result

```

```

# Test
nums = range(1000000)
for n in range(100):

```

```

    r = compute_roots(nums)

```

When tested on our machine, this program runs in about 40 seconds. Now change the `compute_roots()` function as follows:

```

from math import sqrt

```

```

def compute_roots(nums):

```

```

    result = []
    result_append = result.append
    for n in nums:

```

```

        result_append(sqrt(n))

```

```

    return result

```

This version runs in about 29 seconds. The only difference between the two versions of code is the elimination of attribute access. Instead of using `math.sqrt()`, the code uses `sqrt()`. The `result.append()` method is additionally placed into a local variable `result_append` and reused in the inner loop. However, it must be emphasized that these changes only make sense in frequently executed code, such as loops. So, this optimization really only makes sense in carefully selected places.

Understand locality of variables As previously noted, local variables are faster than global variables. For frequently accessed names, speedups can be obtained by making those names as local as possible. For example, consider this modified version of the `compute_roots()` function just discussed:

```

import math

```

```

def compute_roots(nums):

```

```

    sqrt = math.sqrt
    result = []
    result_append = result.append
    for n in nums:

```

```

        result_append(sqrt(n))

```

```

    return result

```

In this version, `sqrt` has been lifted from the `math` module and placed into a local variable. If you run this code, it now runs in about 25 seconds (an improvement over the previous version, which took 29 seconds). That additional speedup is due to a local lookup of `sqrt` being a bit faster than a global lookup of `sqrt`. Locality arguments also apply when working in classes. In general, looking up a value such as `self.name` will be considerably slower than accessing a local variable. In inner loops, it might pay to lift commonly accessed attributes into a local variable. For example:

```
# Slower class SomeClass:
```

```
... def method(self):  
  
    for x in s:  
        op(self.value)
```

```
# Faster class SomeClass:
```

```
... def method(self):  
  
    value = self.value  
    for x in s:  
        op(value)
```

Avoid gratuitous abstraction Any time you wrap up code with extra layers of processing, such as decorators, properties, or descriptors, you're going to make it slower. As an example, consider this class:

```
class A:  
    def __init__(self, x, y):  
        self.x = x self.y = y  
  
    @property def y(self):  
        return self._y  
  
    @y.setter def y(self, value):  
        self._y = value
```

Now, try a simple timing test:


```
>>> from timeit import timeit
>>> a = A(1,2)
>>> timeit('a.x', 'from __main__ import a')
0.07817923510447145
>>> timeit('a.y', 'from __main__ import a')
0.35766440676525235
>>>
```

As you can observe, accessing the property `y` is not just slightly slower than a simple attribute `x`, it's about 4.5 times slower. If this difference matters, you should ask yourself if the definition of `y` as a property was really necessary. If not, simply get rid of it and go back to using a simple attribute instead. Just because it might be common for programs in another programming language to use getter/setter functions, that doesn't mean you should adopt that programming style for Python.

Use the built-in containers Built-in data types such as strings, tuples, lists, sets, and dicts are all implemented in C, and are rather fast. If you're inclined to make your own data structures as a replacement (e.g., linked lists, balanced trees, etc.), it may be rather difficult if not impossible to match the speed of the built-ins. Thus, you're often better off just using them.

Avoid making unnecessary data structures or copies Sometimes programmers get carried away with making unnecessary data structures when they just don't have to. For example, someone might write code like this:

```
values = [x for x in sequence] squares = [x*x for x in values]
```

Perhaps the thinking here is to first collect a bunch of values into a list and then to start applying operations such as list comprehensions to it. However, the first list is completely unnecessary. Simply write the code like this:

```
squares = [x*x for x in sequence]
```

Related to this, be on the lookout for code written by programmers who are overly paranoid about Python's sharing of values. Overuse of functions such as `copy.deepcopy()` may be a sign of code that's been written by someone who doesn't fully understand or trust Python's memory model. In such code, it may be safe to eliminate many of the copies.

讨论

Before optimizing, it's usually worthwhile to study the algorithms that you're using first. You'll get a much bigger speedup by switching to an $O(n \log n)$ algorithm than by trying to tweak the implementation of an $O(n^2)$ algorithm. If you've decided that you still must optimize, it pays to consider the big picture. As a general rule, you don't want to apply optimizations to every part of your program, because such changes are going to make the code hard to read and understand. Instead, focus only on known performance bottlenecks, such as inner loops. You need to be especially wary interpreting the results of micro-optimizations. For example, consider these two techniques for creating a dictionary:

```
a = {
    'name': 'AAPL', 'shares': 100, 'price': 534.22
}
```

```
b = dict(name='AAPL', shares=100, price=534.22)
```

The latter choice has the benefit of less typing (you don't need to quote the key names). However, if you put the two code fragments in a head-to-head performance battle, you'll find that using `dict()` runs three times slower! With this knowledge, you might be inclined to scan your code and replace every use of `dict()` with its more verbose alternative. However, a smart programmer will only focus on parts of a program where it might actually matter, such as an inner loop. In other places, the speed difference just isn't going to matter at all. If, on the other hand, your performance needs go far beyond the simple techniques in this recipe, you might investigate the use of tools based on just-in-time (JIT) compilation techniques. For example, the PyPy project is an alternate implementation of the Python

interpreter that analyzes the execution of your program and generates native machine code for frequently executed parts. It can sometimes make Python programs run an order of magnitude faster, often approaching (or even exceeding) the speed of code written in C. Unfortunately, as of this writing, PyPy does not yet fully support Python 3. So, that is something to look for in the future. You might also consider the Numba project. Numba is a dynamic compiler where you annotate selected Python functions that you want to optimize with a decorator. Those functions are then compiled into native machine code through the use of LLVM. It too can produce significant performance gains. However, like PyPy, support for Python 3 should be viewed as somewhat experimental. Last, but not least, the words of John Ousterhout come to mind: "The best performance improvement is the transition from the nonworking to the working state." Don't worry about optimization until you need to. Making sure your program works correctly is usually more important than making it run fast (at least initially).

第十五章：C语言扩展

本章着眼于从Python访问C代码的问题。许多Python内置库是用C写的，访问C是让Python的对现有库进行交互一个重要的组成部分。这也是一个当你面临从Python 2到Python 3扩展代码的问题。虽然Python提供了一个广泛的编程API，实际上有很多方法来处理C的代码。相比试图给出对于每一个可能的工具或技术的详细参考，我么采用的是集中在一个小片段的C++代码，以及一些有代表性的例子来展示如何与代码交互。这个目标是提供一系列的编程模板，有经验的程序员可以扩展自己的使用。

这里是我们将大部分秘籍中工作的代码：

```

/* sample.c */_method
#include <math.h>

/* Compute the greatest common divisor */
int gcd(int x, int y) {
    int g = y;
    while (x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
    return g;
}

/* Test if (x0,y0) is in the Mandelbrot set or not */
int in_mandel(double x0, double y0, int n) {
    double x=0,y=0,xtemp;
    while (n > 0) {
        xtemp = x*x - y*y + x0;
        y = 2*x*y + y0;
        x = xtemp;
        n -= 1;
        if (x*x + y*y > 4) return 0;
    }
    return 1;
}

/* Divide two numbers */
int divide(int a, int b, int *remainder) {
    int quot = a / b;
    *remainder = a % b;
    return quot;
}

/* Average values in an array */
double avg(double *a, int n) {
    int i;
    double total = 0.0;
    for (i = 0; i < n; i++) {
        total += a[i];
    }
    return total / n;
}

/* A C data structure */
typedef struct Point {
    double x,y;
} Point;

/* Function involving a C data structure */
double distance(Point *p1, Point *p2) {
    return hypot(p1->x - p2->x, p1->y - p2->y);
}

```

This code contains a number of different C programming features. First, there are a few simple functions such as `gcd()` and `is_mandel()`. The `divide()` function is an example of a C function returning multiple values, one through a pointer argument. The `avg()` function performs a data reduction across a C array. The `Point` and `distance()` function involve C structures.

For all of the recipes that follow, assume that the preceding code is found in a file named `sample.c`, that definitions are found in a file named `sample.h` and that it has been compiled into a library `libsampl` that can be linked to other C code. The exact details of compilation and linking vary from system to system, but that is not the primary focus. It is assumed that if you're working with C code, you've already figured that out.

Contents:

15.1 使用 `ctypes` 访问 C 代码

问题

You have a small number of C functions that have been compiled into a shared library or DLL. You would like to call these functions purely from Python without having to write additional C code or using a third-party extension tool.

解决方案

For small problems involving C code, it is often easy enough to use the `ctypes` module that is part of Python's standard library. In order to use `ctypes`, you must first make sure the C code you want to access has been compiled into a shared library that is compatible with the Python interpreter (e.g., same architecture, word size, compiler, etc.). For the purposes of this recipe, assume that a shared library, `libsampl.so`, has been created and that it contains nothing more than the code shown in the chapter introduction. Further assume that the `libsampl.so` file has been placed in the same directory as the `sample.py` file shown next. To access the resulting library, you make a Python module that wraps around it, such as the following: `# sample.py`

```
import os
# Try to locate the .so file in the same directory as this file
_file = 'libsampl.so'
_path = os.path.join(*os.path.split(_file[:-1]) + (_file,))
_mod = ctypes.cdll.LoadLibrary(_path)
```

```
# int gcd(int, int)
gcd = _mod.gcd
gcd.argtypes = (ctypes.c_int, ctypes.c_int)
gcd.restype = ctypes.c_int
```

```
# int in_mandel(double, double, int) in_mandel = _mod.in_mandel in_mandel.argtypes =
(ctypes.c_double, ctypes.c_double, ctypes.c_int) in_mandel.restype = ctypes.c_int
```

```
# int divide(int, int, int *) _divide = _mod.divide _divide.argtypes = (ctypes.c_int, ctypes.c_int,
ctypes.POINTER(ctypes.c_int)) _divide.restype = ctypes.c_int
```

```
def divide(x, y):
```

```
    rem = ctypes.c_int() quot = _divide(x, y, rem)
```

```
    return quot, rem.value
```

```
# void avg(double *, int n) # Define a special type for the 'double *' argument class
DoubleArrayType:
```

```
def from_param(self, param):
```

```
    typename = type(param).__name__ if hasattr(self, 'from_' + typename):
```

```
        return getattr(self, 'from_' + typename)(param)
```

```
elif isinstance(param, ctypes.Array):
```

```
    return param
```

```
else:
```

```
    raise TypeError("Can't convert %s" % typename)
```

```
# Cast from array.array objects def from_array(self, param):
```

```
if param.typecode != 'd':
```

```
    raise TypeError('must be an array of doubles')
```

```
    ptr, _ = param.buffer_info() return ctypes.cast(ptr,
ctypes.POINTER(ctypes.c_double))
```

```
# Cast from lists/tuples def from_list(self, param):
```

```
    val = ((ctypes.c_double)*len(param))(*param) return val
```

```
from_tuple = from_list
```

```
# Cast from a numpy array def from_ndarray(self, param):
```

```
    return param.ctypes.data_as(ctypes.POINTER(ctypes.c_double))
```

```
DoubleArray = DoubleArrayType() _avg = _mod.avg _avg.argtypes = (DoubleArray,
ctypes.c_int) _avg.restype = ctypes.c_double
```

```

def avg(values):
    return _avg(values, len(values))

# struct Point { } class Point(ctypes.Structure):

    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]

# double distance(Point *, Point *) distance = _mod.distance distance.argtypes =
(ctypes.POINTER(Point), ctypes.POINTER(Point)) distance.restype = ctypes.c_double

```

If all goes well, you should be able to load the module and use the resulting C functions. For example:

```

>>> import sample
>>> sample.gcd(35,42)
7
>>> sample.in_mandel(0,0,500)
1
>>> sample.in_mandel(2.0,1.0,500)
0
>>> sample.divide(42,8)
(5, 2)
>>> sample.avg([1,2,3])
2.0
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
4.242640687119285
>>>

```

讨论

There are several aspects of this recipe that warrant some discussion. The first issue concerns the overall packaging of C and Python code together. If you are using ctypes to access C code that you have compiled yourself, you will need to make sure that the shared library gets placed in a location where the sample.py module can find it. One possibility is to put the resulting .so file in the same directory as the supporting Python code. This is what's shown at the first part of this recipe—sample.py looks at the `__file__` variable to see where it has been installed, and then constructs a path that points to a `libsampl.so` file in the same directory. If the C library is going to be installed elsewhere, then you'll have to adjust the path accordingly. If the C library is installed as a standard library on your machine, you might be able to use the `ctypes.util.find_library()` function. For example:

```
>>> from ctypes.util import find_library
>>> find_library('m')
'/usr/lib/libm.dylib'
>>> find_library('pthread')
'/usr/lib/libpthread.dylib'
>>> find_library('sample')
'/usr/local/lib/libsample.so'
>>>
```

Again, ctypes won't work at all if it can't locate the library with the C code. Thus, you'll need to spend a few minutes thinking about how you want to install things. Once you know where the C library is located, you use `ctypes.cdll.LoadLibrary()` to load it. The following statement in the solution does this where `_path` is the full pathname to the shared library:

```
_mod = ctypes.cdll.LoadLibrary(_path)
```

Once a library has been loaded, you need to write statements that extract specific symbols and put type signatures on them. This is what's happening in code fragments such as this:

```
# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int
```

In this code, the `.argtypes` attribute is a tuple containing the input arguments to a function, and `.restype` is the return type. ctypes defines a variety of type objects (e.g., `c_double`, `c_int`, `c_short`, `c_float`, etc.) that represent common C data types. Attaching the type signatures is critical if you want to make Python pass the right kinds of arguments and convert data correctly (if you don't do this, not only will the code not work, but you might cause the entire interpreter process to crash). A somewhat tricky part of using ctypes is that the original C code may use idioms that don't map cleanly to Python. The `divide()` function is a good example because it returns a value through one of its arguments. Although that's a common C technique, it's often not clear how it's supposed to work in Python. For example, you can't do anything straightforward like this:

```
>>> divide = _mod.divide
>>> divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
>>> x = 0
>>> divide(10, 3, x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 3: <class 'TypeError'>: expected LP_c_int
instance instead of int
>>>
```


Even if this did work, it would violate Python's immutability of integers and probably cause the entire interpreter to be sucked into a black hole. For arguments involving pointers, you usually have to construct a compatible ctypes object and pass it in like this:

```
>>> x = ctypes.c_int()
>>> divide(10, 3, x)
3
>>> x.value
1
>>>
```

Here an instance of a `ctypes.c_int` is created and passed in as the pointer object. Unlike a normal Python integer, a `c_int` object can be mutated. The `.value` attribute can be used to either retrieve or change the value as desired.

For cases where the C calling convention is “un-Pythonic,” it is common to write a small wrapper function. In the solution, this code makes the `divide()` function return the two results using a tuple instead: `# int divide(int, int, int *) _divide = _mod.divide _divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int)) _divide.restype = ctypes.c_int`

```
def divide(x, y):
    rem = ctypes.c_int() quot = _divide(x,y,rem) return quot, rem.value
```

The `avg()` function presents a new kind of challenge. The underlying C code expects to receive a pointer and a length representing an array. However, from the Python side, we must consider the following questions: What is an array? Is it a list? A tuple? An array from the `array` module? A `numpy` array? Is it all of these? In practice, a Python “array” could take many different forms, and maybe you would like to support multiple possibilities. The `DoubleArrayType` class shows how to handle this situation. In this class, a single method `from_param()` is defined. The role of this method is to take a single parameter and narrow it down to a compatible ctypes object (a pointer to a `ctypes.c_double`, in the example). Within `from_param()`, you are free to do anything that you wish. In the solution, the `typename` of the parameter is extracted and used to dispatch to a more specialized method. For example, if a list is passed, the `typename` is `list` and a method `from_list()` is invoked. For lists and tuples, the `from_list()` method performs a conversion to a ctypes array object. This looks a little weird, but here is an interactive example of converting a list to a ctypes array:

```

>>> nums = [1, 2, 3]
>>> a = (ctypes.c_double * len(nums))(*nums)
>>> a
<__main__.c_double_Array_3 object at 0x10069cd40>
>>> a[0]
1.0
>>> a[1]
2.0
>>> a[2]
3.0
>>>

```

For array objects, the `from_array()` method extracts the underlying memory pointer and casts it to a ctypes pointer object. For example:

```

>>> import array
>>> a = array.array('d', [1,2,3])
>>> a
array('d', [1.0, 2.0, 3.0])
>>> ptr_ = a.buffer_info()
>>> ptr
4298687200
>>> ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
<__main__.LP_c_double object at 0x10069cd40>
>>>

```

The `from_ndarray()` shows comparable conversion code for numpy arrays. By defining the `DoubleArrayType` class and using it in the type signature of `avg()`, as shown, the function can accept a variety of different array-like inputs:

```

>>> import sample
>>> sample.avg([1,2,3])
2.0
>>> sample.avg((1,2,3))
2.0
>>> import array
>>> sample.avg(array.array('d', [1,2,3]))
2.0
>>> import numpy
>>> sample.avg(numpy.array([1.0,2.0,3.0]))
2.0
>>>

```

The last part of this recipe shows how to work with a simple C structure. For structures, you simply define a class that contains the appropriate fields and types like this:

```
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]
```

Once defined, you can use the class in type signatures as well as in code that needs to instantiate and work with the structures. For example:

```
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> p1.x
1.0
>>> p1.y
2.0
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

A few final comments: ctypes is a useful library to know about if all you're doing is accessing a few C functions from Python. However, if you're trying to access a large library, you might want to look at alternative approaches, such as Swig (described in Recipe 15.9) or Cython (described in Recipe 15.10).

The main problem with a large library is that since ctypes isn't entirely automatic, you'll have to spend a fair bit of time writing out all of the type signatures, as shown in the example. Depending on the complexity of the library, you might also have to write a large number of small wrapper functions and supporting classes. Also, unless you fully understand all of the low-level details of the C interface, including memory management and error handling, it is often quite easy to make Python catastrophically crash with a segmentation fault, access violation, or some similar error. As an alternative to ctypes, you might also look at CFFI. CFFI provides much of the same functionality, but uses C syntax and supports more advanced kinds of C code. As of this writing, CFFI is still a relatively new project, but its use has been growing rapidly. There has even been some discussion of including it in the Python standard library in some future release. Thus, it's definitely something to keep an eye on.

15.2 简单的C扩展模块

问题

You want to write a simple C extension module directly using Python's extension API and no other tools.

解决方案

For simple C code, it is straightforward to make a handcrafted extension module. As a preliminary step, you probably want to make sure your C code has a proper header file. For example,

```
/* sample.h */
```

```
#include <math.h>
```

```
extern int gcd(int, int); extern int in_mandel(double x0, double y0, int n); extern int  
divide(int a, int b, int *remainder); extern double avg(double *a, int n);
```

```
typedef struct Point {  
    double x,y;
```

```
} Point;
```

```
extern double distance(Point *p1, Point *p2);
```

Typically, this header would correspond to a library that has been compiled separately. With that assumption, here is a sample extension module that illustrates the basics of writing extension functions:

```
#include "Python.h" #include "sample.h"
```

```
/* int gcd(int, int) */ static PyObject *py_gcd(PyObject *self, PyObject *args) {
```

```
    int x, y, result;
```

```
    if (!PyArg_ParseTuple(args, "ii", &x, &y)) {  
        return NULL;
```

```
    } result = gcd(x,y); return Py_BuildValue("i", result);
```

```
}
```

```
/* int in_mandel(double, double, int) */ static PyObject *py_in_mandel(PyObject *self,  
PyObject *args) {
```

```

double x0, y0; int n; int result;

if (!PyArg_ParseTuple(args, "ddi", &x0, &y0, &n)) {
    return NULL;

} result = in_mandel(x0,y0,n); return Py_BuildValue("i", result);

}

/* int divide(int, int, int *) */ static PyObject *py_divide(PyObject *self, PyObject *args) {

    int a, b, quotient, remainder; if (!PyArg_ParseTuple(args, "ii", &a, &b)) {

        return NULL;

    } quotient = divide(a,b, &remainder); return Py_BuildValue("(ii)", quotient, remainder);

}

/* Module method table */ static PyMethodDef SampleMethods[] = {

    {"gcd", py_gcd, METH_VARARGS, "Greatest common divisor"}, {"in_mandel",
    py_in_mandel, METH_VARARGS, "Mandelbrot test"}, {"divide", py_divide,
    METH_VARARGS, "Integer division"}, { NULL, NULL, 0, NULL}

};

/* Module structure */ static struct PyModuleDef samplemodule = {

    PyModuleDef_HEAD_INIT,

    "sample", /* name of module / "A sample module", / Doc string (may be NULL) / -1, / Size
    of per-interpreter state or -1 / SampleMethods / Method table */

};

/* Module initialization function */ PyMODINIT_FUNC PyInit_sample(void) {

    return PyModule_Create(&samplemodule);

}

```

For building the extension module, create a setup.py file that looks like this:

```
# setup.py from distutils.core import setup, Extension
```

```
setup(name='sample',
      ext_modules=[
          Extension('sample',
                  ['pysample.c'], include_dirs = ['/some/dir'], define_macros = [('FOO', '1')],
                  undef_macros = ['BAR'], library_dirs = ['/usr/local/lib'], libraries = ['sample'])
      ]
)
```

Now, to build the resulting library, simply use `python3 buildlib.py build_ext - inplace`. For example:

```
bash % python3 setup.py build_ext -inplace running build_ext building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
```

```
-I/usr/local/include/python3.3m -c pysample.c -o build/temp.macosx-10.6-x86_64-3.3/pysample.o
```

```
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/pysample.o
```

```
-L/usr/local/lib -lsample -o sample.so
```

```
bash %
```

As shown, this creates a shared library called `sample.so`. When compiled, you should be able to start importing it as a module:

```
>>> import sample
>>> sample.gcd(35, 42)
7
>>> sample.in_mandel(0, 0, 500)
1
>>> sample.in_mandel(2.0, 1.0, 500)
```

```
0 >>> sample.divide(42, 8) (5, 2) >>>
```

If you are attempting these steps on Windows, you may need to spend some time fiddling with your environment and the build environment to get extension modules to build correctly. Binary distributions of Python are typically built using Microsoft Visual Studio. To get extensions to work, you may have to compile them using the same or compatible tools. See the Python documentation.

讨论

Before attempting any kind of handwritten extension, it is absolutely critical that you consult Python's documentation on "Extending and Embedding the Python Interpreter". Python's C extension API is large, and repeating all of it here is simply not practical. However, the most important parts can be easily discussed. First, in extension modules, functions that you write are all typically written with a common prototype such as this:

```
static PyObject *py_func(PyObject *self, PyObject *args) {  
    ...  
}
```

PyObject is the C data type that represents any Python object. At a very high level, an extension function is a C function that receives a tuple of Python objects (in PyObject *args) and returns a new Python object as a result. The self argument to the function is unused for simple extension functions, but comes into play should you want to define new classes or object types in C (e.g., if the extension function were a method of a class, then self would hold the instance). The PyArg_ParseTuple() function is used to convert values from Python to a C representation. As input, it takes a format string that indicates the required values, such as "i" for integer and "d" for double, as well as the addresses of C variables in which to place the converted results. PyArg_ParseTuple() performs a variety of checks on the number and type of arguments. If there is any mismatch with the format string, an exception is raised and NULL is returned. By checking for this and simply returning NULL, an appropriate exception will have been raised in the calling code. The Py_BuildValue() function is used to create Python objects from C data types. It also accepts a format code to indicate the desired type. In the extension functions, it is used to return results back to Python. One feature of Py_BuildValue() is that it can build more complicated kinds of objects, such as tuples and dictionaries. In the code for py_divide(), an example showing the return of a tuple is shown. However, here are a few more examples:

```
return Py_BuildValue("i", 34); // Return an integer  
return Py_BuildValue("d", 3.4); // Return  
a double  
return Py_BuildValue("s", "Hello"); // Null-terminated UTF-8 string  
return  
Py_BuildValue("(ii)", 3, 4); // Tuple (3, 4)
```

Near the bottom of any extension module, you will find a function table such as the SampleMethods table shown in this recipe. This table lists C functions, the names to use in Python, as well as doc strings. All modules are required to specify such a table, as it gets used in the initialization of the module. The final function PyInit_sample() is the module initialization function that executes when the module is first imported. The primary job of this function is to register the module object with the interpreter. As a final note, it must be stressed that there is considerably more to extending Python with C functions than what is

shown here (in fact, the C API contains well over 500 functions in it). You should view this recipe simply as a stepping stone for getting started. To do more, start with the documentation on the `PyArg_ParseTuple()` and `Py_BuildValue()` functions, and expand from there.

15.3 一个操作数组的扩展函数

问题

You want to write a C extension function that operates on contiguous arrays of data, as might be created by the array module or libraries like NumPy. However, you would like your function to be general purpose and not specific to any one array library.

解决方案

To receive and process arrays in a portable manner, you should write code that uses the Buffer Protocol. Here is an example of a handwritten C extension function that receives array data and calls the `avg(double *buf, int len)` function from this chapter's introduction:

```
/* Call double avg(double *, int) */ static PyObject *py_avg(PyObject *self, PyObject *args) {
```



```
PyObject bufobj; Py_buffer view; double result; /* Get the passed Python object */ if (!PyArg_ParseTuple(args, "O", &bufobj)) {
```

```
    return NULL;
```

```
}
```

```
/* Attempt to extract buffer information from it */
```

```
if (PyObject_GetBuffer(bufobj, &view,  
    PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
```

```
    return NULL;
```

```
}
```

```
if (view.ndim != 1) {
```

```
    PyErr_SetString(PyExc_TypeError, "Expected a 1-dimensional array");
```

```
    PyBuffer_Release(&view); return NULL;
```

```
}
```

```
/* Check the type of items in the array */ if (strcmp(view.format, "d") != 0) {
```

```
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
```

```
    PyBuffer_Release(&view); return NULL;
```

```
}
```

```
/* Pass the raw buffer and size to the C function */ result = avg(view.buf, view.shape[0]);
```

```
/* Indicate we're done working with the buffer */ PyBuffer_Release(&view); return  
Py_BuildValue("d", result);
```

```
}
```

Here is an example that shows how this extension function works:

```

>>> import array
>>> avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> avg(numpy.array([1.0,2.0,3.0]))
2.0
>>> avg([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' does not support the buffer interface
>>> avg(b'Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected an array of doubles
>>> a = numpy.array([[1.,2.,3.],[4.,5.,6.]])
>>> avg(a[:,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ndarray is not contiguous
>>> sample.avg(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected a 1-dimensional array
>>> sample.avg(a[0])

```

2.0

讨论

Passing array objects to C functions might be one of the most common things you would want to do with a extension function. A large number of Python applications, ranging from image processing to scientific computing, are based on high-performance array processing. By writing code that can accept and operate on arrays, you can write customized code that plays nicely with those applications as opposed to having some sort of custom solution that only works with your own code. The key to this code is the `PyBuffer_GetBuffer()` function. Given an arbitrary Python object, it tries to obtain information about the underlying memory representation. If it's not possible, as is the case with most normal Python objects, it simply raises an exception and returns -1. The special flags passed to `PyBuffer_GetBuffer()` give additional hints about the kind of memory buffer that is requested. For example, `PyBUF_ANY_CONTIGUOUS` specifies that a contiguous region of memory is required. For arrays, byte strings, and other similar objects, a `Py_buffer` structure is filled with information about the underlying memory. This includes a pointer to the memory, size, itemsize, format, and other details. Here is the definition of this structure:

```

typedef struct bufferinfo {
    void buf; / Pointer to buffer memory / PyObject *obj; / Python object that is the owner /

```

```
Py_ssize_t len; / Total size in bytes / Py_ssize_t itemsize; / Size in bytes of a single item /
int readonly; / Read-only access flag / int ndim; / Number of dimensions / char *format; /
struct code of a single item / Py_ssize_t *shape; / Array containing dimensions /
Py_ssize_t *strides; / Array containing strides / Py_ssize_t *suboffsets; / Array containing
suboffsets */
```

```
} Py_buffer;
```

In this recipe, we are simply concerned with receiving a contiguous array of doubles. To check if items are a double, the format attribute is checked to see if the string is “d”. This is the same code that the struct module uses when encoding binary values. As a general rule, format could be any format string that’s compatible with the struct module and might include multiple items in the case of arrays containing C structures. Once we have verified the underlying buffer information, we simply pass it to the C function, which treats it as a normal C array. For all practical purposes, it is not concerned with what kind of array it is or what library created it. This is how the function is able to work with arrays created by the array module or by numpy.

Before returning a final result, the underlying buffer view must be released using `PyBuffer_Release()`. This step is required to properly manage reference counts of objects. Again, this recipe only shows a tiny fragment of code that receives an array. If working with arrays, you might run into issues with multidimensional data, strided data, different data types, and more that will require study. Make sure you consult the official documentation to get more details. If you need to write many extensions involving array handling, you may find it easier to implement the code in Cython. See Recipe 15.11.

15.4 在C扩展模块中操作隐形指针

问题

You have an extension module that needs to handle a pointer to a C data structure, but you don’t want to expose any internal details of the structure to Python.

解决方案

Opaque data structures are easily handled by wrapping them inside capsule objects. Consider this fragment of C code from our sample code:

```
typedef struct Point {
    double x,y;
```

```
} Point;
```

```
extern double distance(Point *p1, Point *p2);
```

Here is an example of extension code that wraps the Point structure and distance() function using capsules:

```
/* Destructor function for points */ static void del_Point(PyObject *obj) {
```

```
    free(PyCapsule_GetPointer(obj,"Point"));
```

```
}
```

```
/* Utility functions */ static Point *PyPoint_AsPoint(PyObject *obj) {
```

```
    return (Point *) PyCapsule_GetPointer(obj, "Point");
```

```
}
```

```
static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
```

```
    return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
```

```
}
```

```
/* Create a new Point object */ static PyObject *py_Point(PyObject *self, PyObject *args) {
```

```
    Point *p; double x,y; if (!PyArg_ParseTuple(args,"dd",&x,&y)) {
```

```
        return NULL;
```

```
    } p = (Point *) malloc(sizeof(Point)); p->x = x; p->y = y; return PyPoint_FromPoint(p, 1);
```

```
}
```

```
static PyObject *py_distance(PyObject *self, PyObject *args) {
```

```
    Point *p1, *p2; PyObject *py_p1, *py_p2; double result;
```

```
    if (!PyArg_ParseTuple(args,"OO",&py_p1, &py_p2)) {
```

```
        return NULL;
```

```
    } if (!(p1 = PyPoint_AsPoint(py_p1))) {
```

```
        return NULL;
```

```
    } if (!(p2 = PyPoint_AsPoint(py_p2))) {
```

```
        return NULL;
```

```
    } result = distance(p1,p2); return Py_BuildValue("d", result);  
}
```

Using these functions from Python looks like this:

```
>>> import sample  
>>> p1 = sample.Point(2,3)  
>>> p2 = sample.Point(4,5)  
>>> p1  
<capsule object "Point" at 0x1004ea330>  
>>> p2  
<capsule object "Point" at 0x1005d1db0>  
>>> sample.distance(p1,p2)  
2.8284271247461903  
>>>
```

讨论

Capsules are similar to a typed C pointer. Internally, they hold a generic pointer along with an identifying name and can be easily created using the `PyCapsule_New()` function. In addition, an optional destructor function can be attached to a capsule to release the underlying memory when the capsule object is garbage collected.

To extract the pointer contained inside a capsule, use the `PyCapsule_GetPointer()` function and specify the name. If the supplied name doesn't match that of the capsule or some other error occurs, an exception is raised and `NULL` is returned. In this recipe, a pair of utility functions—`PyPoint_FromPoint()` and `PyPoint_AsPoint()`—have been written to deal with the mechanics of creating and unwinding `Point` instances from capsule objects. In any extension functions, we'll use these functions instead of working with capsules directly. This design choice makes it easier to deal with possible changes to the wrapping of `Point` objects in the future. For example, if you decided to use something other than a capsule later, you would only have to change these two functions. One tricky part about capsules concerns garbage collection and memory management. The `PyPoint_FromPoint()` function accepts a `must_free` argument that indicates whether the underlying `Point` structure is to be collected when the capsule is destroyed. When working with certain kinds of C code, ownership issues can be difficult to handle (e.g., perhaps a `Point` structure is embedded within a larger data structure that is managed separately). Rather than making a unilateral decision to garbage collect, this extra argument gives control back to the programmer. It should be noted that the destructor associated with an existing capsule can also be changed using the `PyCapsule_SetDestructor()` function. Capsules are a sensible solution to interfacing with certain kinds of C code involving structures. For instance, sometimes you

just don't care about exposing the internals of a structure or turning it into a full-fledged extension type. With a capsule, you can put a lightweight wrapper around it and easily pass it around to other extension functions.

15.5 从扩张模块中定义和导出C的API

问题

You have a C extension module that internally defines a variety of useful functions that you would like to export as a public C API for use elsewhere. You would like to use these functions inside other extension modules, but don't know how to link them together, and doing it with the C compiler/linker seems excessively complicated (or impossible).

解决方案

This recipe focuses on the code written to handle Point objects, which were presented in Recipe 15.4. If you recall, that C code included some utility functions like this:

```
/* Destructor function for points */ static void del_Point(PyObject *obj) {  
  
    free(PyCapsule_GetPointer(obj,"Point"));  
  
}  
  
/* Utility functions */ static Point *PyPoint_AsPoint(PyObject *obj) {  
  
    return (Point *) PyCapsule_GetPointer(obj, "Point");  
  
}  
  
static PyObject *PyPoint_FromPoint(Point *p, int must_free) {  
    return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);  
  
}
```

The problem now addressed is how to export the PyPoint_AsPoint() and PyPoint_FromPoint() functions as an API that other extension modules could use and link to (e.g., if you have other extensions that also want to use the wrapped Point objects). To solve this problem, start by introducing a new header file for the "sample" extension called pysample.h. Put the following code in it:

```

/* pysample.h */ #include "Python.h" #include "sample.h" #ifdef __cplusplus extern "C" {
#endif

/* Public API Table */ typedef struct {

    Point *(*aspoint)(PyObject *); PyObject *(*frompoint)(Point *, int);

} _PointAPIMethods;

#ifdef PYSAMPLE_MODULE /* Method table in external module */ static
_point_api _point_api = 0;

/* Import the API table from sample */ static int import_sample(void) {

    _point_api = (_PointAPIMethods *) PyCapsule_Import("sample._point_api",0); return
    (_point_api != NULL) ? 1 : 0;

}

/* Macros to implement the programming interface */ #define PyPoint_AsPoint(obj)
(_point_api->aspoint)(obj) #define PyPoint_FromPoint(obj) (_point_api->frompoint)(obj)
#endif

#ifdef __cplusplus } #endif

```

The most important feature here is the `_PointAPIMethods` table of function pointers. It will be initialized in the exporting module and found by importing modules. Change the original extension module to populate the table and export it as follows:

```

/* pysample.c */

#include "Python.h" #define PYSAMPLE_MODULE #include "pysample.h"

... /* Destructor function for points */ static void del_Point(PyObject *obj) {

    printf("Deleting point\n"); free(PyCapsule_GetPointer(obj,"Point"));

}

/* Utility functions */ static Point *PyPoint_AsPoint(PyObject *obj) {

    return (Point *) PyCapsule_GetPointer(obj,"Point");
}

```

```

}

static PyObject *PyPoint_FromPoint(Point *p, int free) {
    return PyCapsule_New(p, "Point", free ? del_Point : NULL);
}

static _PointAPIMethods _point_api = {
    PyPoint_AsPoint, PyPoint_FromPoint
};

/* Module initialization function */ PyMODINIT_FUNC PyInit_sample(void) {

    PyObject *m; PyObject *py_point_api;

    m = PyModule_Create(&samplemodule); if (m == NULL)

        return NULL;

    /* Add the Point C API functions */ py_point_api = PyCapsule_New((void *) &_point_api,
    "sample._point_api", NULL); if (py_point_api) {

        PyModule_AddObject(m, "_point_api", py_point_api);

    } return m;
}

```

Finally, here is an example of a new extension module that loads and uses these API functions:

```

/* pexample.c */

/* Include the header associated with the other module */ #include "pysample.h"

/* An extension function that uses the exported API */ static PyObject
*print_point(PyObject *self, PyObject *args) {

```



```
PyObject *obj; Point *p; if (!PyArg_ParseTuple(args, "O", &obj)) {
```

```
    return NULL;
```

```
}
```

```
/* Note: This is defined in a different module */ p = PyPoint_AsPoint(obj); if (!p) {
```

```
    return NULL;
```

```
} printf("%f %fn", p->x, p->y); return Py_BuildValue("");
```

```
}
```

```
static PyMethodDef PtExampleMethods[] = {
```

```
    {"print_point", print_point, METH_VARARGS, "output a point"}, { NULL, NULL, 0, NULL}
```

```
};
```

```
static struct PyModuleDef ptexamplemodule = {
```

```
    PyModuleDef_HEAD_INIT, "ptexample", /* name of module / "A module that imports an
```

```
    API", / Doc string (may be NULL) / -1, / Size of per-interpreter state or -1 /
```

```
    PtExampleMethods / Method table */
```

```
};
```

```
/* Module initialization function */ PyMODINIT_FUNC PyInit_ptexample(void) {
```

```
    PyObject *m;
```

```
    m = PyModule_Create(&ptexamplemodule); if (m == NULL)
```

```
        return NULL;
```

```
/* Import sample, loading its API functions */ if (!import_sample()) {
```

```
    return NULL;
```

```
}
```

```
return m;
```

```
}
```

When compiling this new module, you don't even need to bother to link against any of the libraries or code from the other module. For example, you can just make a simple setup.py file like this:

```
# setup.py from distutils.core import setup, Extension
```

```
setup(name='ptexample',
      ext_modules=[
          Extension('ptexample',
                  ['ptexample.c'], include_dirs = [], # May need pysample.h directory)
      ]
)
```

If it all works, you'll find that your new extension function works perfectly with the C API functions defined in the other module:

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p1
<capsule object "Point *" at 0x1004ea330>
>>> import ptexample
>>> ptexample.print_point(p1)
2.000000 3.000000
>>>
```

讨论

This recipe relies on the fact that capsule objects can hold a pointer to anything you wish. In this case, the defining module populates a structure of function pointers, creates a capsule that points to it, and saves the capsule in a module-level attribute (e.g., `sample._point_api`). Other modules can be programmed to pick up this attribute when imported and extract the underlying pointer. In fact, Python provides the `PyCapsule_Import()` utility function, which takes care of all the steps for you. You simply give it the name of the attribute (e.g., `sample._point_api`), and it will find the capsule and extract the pointer all in one step. There are some C programming tricks involved in making exported functions look normal in other modules. In the `pysample.h` file, a pointer `_point_api` is used to point to the method table that was initialized in the exporting module. A related function `import_sample()` is used to perform the required capsule import and initialize this pointer. This function must be called before any functions are used. Normally, it would

be called in during module initialization. Finally, a set of C preprocessor macros have been defined to transparently dispatch the API functions through the method table. The user just uses the original function names, but doesn't know about the extra indirection through these macros. Finally, there is another important reason why you might use this technique to link modules together—it's actually easier and it keeps modules more cleanly decoupled. If you didn't want to use this recipe as shown, you might be able to cross-link modules using advanced features of shared libraries and the dynamic loader. For example, putting common API functions into a shared library and making sure that all extension modules link against that shared library. Yes, this works, but it can be tremendously messy in large systems. Essentially, this recipe cuts out all of that magic and allows modules to link to one another through Python's normal import mechanism and just a tiny number of capsule calls. For compilation of modules, you only need to worry about header files, not the hairy details of shared libraries. Further information about providing C APIs for extension modules can be found in the Python documentation.

15.6 从C语言中调用Python代码

问题

You want to safely execute a Python callable from C and return a result back to C. For example, perhaps you are writing C code that wants to use a Python function as a callback.

解决方案

Calling Python from C is mostly straightforward, but involves a number of tricky parts. The following C code shows an example of how to do it safely:

```
#include <Python.h>

/* Execute func(x,y) in the Python interpreter. The
   arguments and return result of the function must be Python floats */
double call_func(PyObject *func, double x, double y) {
    PyObject *args; PyObject *kwargs; PyObject *result = 0; double retval;

    /* Make sure we own the GIL */ PyGILState_STATE state = PyGILState_Ensure();

    /* Verify that func is a proper callable */ if (!PyCallable_Check(func)) {
        fprintf(stderr, "call_func: expected a callable\n"); goto fail;
    }

    /* Build arguments */ args = Py_BuildValue("(dd)", x, y); kwargs = NULL;
```

```

/* Call the function */ result = PyObject_Call(func, args, kwargs); Py_DECREF(args);
Py_XDECREF(kwargs);

/* Check for Python exceptions (if any) */ if (PyErr_Occurred()) {
    PyErr_Print(); goto fail;
}

/* Verify the result is a float object */ if (!PyFloat_Check(result)) {
    fprintf(stderr, "call_func: callable didn't return a floatn"); goto fail;
}

/* Create the return value */ retval = PyFloat_AsDouble(result); Py_DECREF(result);

/* Restore previous GIL state and return */ PyGILState_Release(state); return retval;

fail:
    Py_XDECREF(result); PyGILState_Release(state); abort(); // Change to something more
    appropriate
}

```

To use this function, you need to have obtained a reference to an existing Python callable to pass in. There are many ways that you can go about doing that, such as having a callable object passed into an extension module or simply writing C code to extract a symbol from an existing module. Here is a simple example that shows calling a function from an embedded Python interpreter:

```

#include <Python.h>

/* Definition of call_func() same as above */ ...

/* Load a symbol from a module */ PyObject *import_name(const char *modname, const
char *symbol) {

    PyObject *u_name, *module; u_name = PyUnicode_FromString(modname); module =
    PyImport_Import(u_name); Py_DECREF(u_name); return
    PyObject_GetAttrString(module, symbol);
}

/* Simple embedding example */ int main() {

```

```

PyObject *pow_func; double x;

Py_Initialize(); /* Get a reference to the math.pow function */ pow_func =
import_name("math", "pow");

/* Call it using our call_func() code */ for (x = 0.0; x < 10.0; x += 0.1) {

    printf("%0.2f %0.2fn", x, call_func(pow_func,x,2.0));

} /* Done */ Py_DECREF(pow_func); Py_Finalize(); return 0;
}

```

To build this last example, you'll need to compile the C and link against the Python interpreter. Here is a Makefile that shows how you might do it (this is something that might require some amount of fiddling with on your machine):

```

all:
    cc -g embed.c -I/usr/local/include/python3.3m
        -L/usr/local/lib/python3.3/config-3.3m -lpython3.3m

```

Compiling and running the resulting executable should produce output similar to this:

```
0.00 0.00 0.10 0.01 0.20 0.04 0.30 0.09 0.40 0.16 ...
```

Here is a slightly different example that shows an extension function that receives a callable and some arguments and passes them to call_func() for the purposes of testing:

```

/* Extension function for testing the C-Python callback */ PyObject *py_call_func(PyObject
*self, PyObject *args) {

    PyObject *func;

    double x, y, result; if (!PyArg_ParseTuple(args, "Odd", &func, &x, &y)) {

        return NULL;

    } result = call_func(func, x, y); return Py_BuildValue("d", result);

}

```

Using this extension function, you could test it as follows:

```

>>> import sample
>>> def add(x,y):
...     return x+y
...
>>> sample.call_func(add,3,4)
7.0
>>>

```

讨论

If you are calling Python from C, the most important thing to keep in mind is that C is generally going to be in charge. That is, C has the responsibility of creating the arguments, calling the Python function, checking for exceptions, checking types, extracting return values, and more. As a first step, it is critical that you have a Python object representing the callable that you're going to invoke. This could be a function, class, method, built-in method, or anything that implements the `__call__()` operation. To verify that it's callable, use `PyCallable_Check()` as shown in this code fragment:

```

double call_func(PyObject *func, double x, double y) {
    .../* Verify that func is a proper callable */ if (!PyCallable_Check(func)) {

        fprintf(stderr, "call_func: expected a callable\n"); goto fail;
    }
}

```

As an aside, handling errors in the C code is something that you will need to carefully study. As a general rule, you can't just raise a Python exception. Instead, errors will have to be handled in some other manner that makes sense to your C code. In the solution, we're using `goto` to transfer control to an error handling block that calls `abort()`. This causes the whole program to die, but in real code you would probably want to do something more graceful (e.g., return a status code). Keep in mind that C is in charge here, so there isn't anything comparable to just raising an exception. Error handling is something you'll have to engineer into the program somehow. Calling a function is relatively straightforward—simply use `PyObject_Call()`, supplying it with the callable object, a tuple of arguments, and an optional dictionary of

keyword arguments. To build the argument tuple or dictionary, you can use `Py_BuildValue()`, as shown.

```

double call_func(PyObject *func, double x, double y) {
    PyObject *args; PyObject *kwargs;

    .../* Build arguments */ args = Py_BuildValue("(dd)", x, y); kwargs = NULL;
}

```

```
/* Call the function */ result = PyObject_Call(func, args, kwargs); Py_DECREF(args);  
Py_XDECREF(kwargs); ...
```

If there are no keyword arguments, you can pass NULL, as shown. After making the function call, you need to make sure that you clean up the arguments using `Py_DECREF()` or `Py_XDECREF()`. The latter function safely allows the NULL pointer to be passed (which is ignored), which is why we're using it for cleaning up the optional keyword arguments. After calling the Python function, you must check for the presence of exceptions. The `PyErr_Occurred()` function can be used to do this. Knowing what to do in response to an exception is tricky. Since you're working from C, you really don't have the exception machinery that Python has. Thus, you would have to set an error status code, log the error, or do some kind of sensible processing. In the solution, `abort()` is called for lack of a simpler alternative (besides, hardened C programmers will appreciate the abrupt crash):

```
... /* Check for Python exceptions (if any) */ if (PyErr_Occurred()) {  
  
    PyErr_Print(); goto fail;
```

fail:

```
    PyGILState_Release(state); abort();
```

Extracting information from the return value of calling a Python function is typically going to involve some kind of type checking and value extraction. To do this, you may have to use functions in the Python concrete objects layer. In the solution, the code checks for and extracts the value of a Python float using `PyFloat_Check()` and `PyFloat_AsDouble()`.

A final tricky part of calling into Python from C concerns the management of Python's global interpreter lock (GIL). Whenever Python is accessed from C, you need to make sure that the GIL is properly acquired and released. Otherwise, you run the risk of having the interpreter corrupt data or crash. The calls to `PyGILState_Ensure()` and `PyGILState_Release()` make sure that it's done correctly:

```
double call_func(PyObject *func, double x, double y) {  
    ... double retval;
```

```
    /* Make sure we own the GIL / PyGILState_STATE state = PyGILState_Ensure(); ... / Code  
    that uses Python C API functions / ... / Restore previous GIL state and return */  
    PyGILState_Release(state); return retval;
```

fail:

```
    PyGILState_Release(state); abort();
```

```
}
```

Upon return, `PyGILState_Ensure()` always guarantees that the calling thread has exclusive access to the Python interpreter. This is true even if the calling C code is running a different thread that is unknown to the interpreter. At this point, the C code is free to use any Python C-API functions that it wants. Upon successful completion, `PyGILState_Release()` is used to restore the interpreter back to its original state. It is critical to note that every `PyGILState_Ensure()` call must be followed by a matching `PyGILState_Release()` call—even in cases where errors have occurred. In the solution, the use of a `goto` statement might look like a horrible design, but we’re actually using it to transfer control to a common exit block that performs this required step. Think of the code after the `fail:` label as serving the same purpose as code in a Python `finally:` block. If you write your C code using all of these conventions including management of the GIL, checking for exceptions, and thorough error checking, you’ll find that you can reliably call into the Python interpreter from C—even in very complicated programs that utilize advanced programming techniques such as multithreading.

15.7 从C扩展中释放全局锁

问题

You have C extension code in that you want to execute concurrently with other threads in the Python interpreter. To do this, you need to release and reacquire the global interpreter lock (GIL).

解决方案

In C extension code, the GIL can be released and reacquired by inserting the following macros in the code:

```
#include "Python.h" ...
```

```
PyObject *pyfunc(PyObject *self, PyObject *args) {  
    ... Py_BEGIN_ALLOW_THREADS // Threaded C code. Must not use Python API  
    functions ... Py_END_ALLOW_THREADS ... return result;  
}
```

讨论

The GIL can only safely be released if you can guarantee that no Python C API functions will be executed in the C code. Typical examples where the GIL might be released are in computationally intensive code that performs calculations on C arrays (e.g., in extensions such as numpy) or in code where blocking I/O operations are going to be performed (e.g., reading or writing on a file descriptor). While the GIL is released, other Python threads are allowed to execute in the interpreter. The `Py_END_ALLOW_THREADS` macro blocks execution until the calling threads reacquires the GIL in the interpreter.

15.8 C和Python中的线程混用

问题

You have a program that involves a mix of C, Python, and threads, but some of the threads are created from C outside the control of the Python interpreter. Moreover, certain threads utilize functions in the Python C API.

解决方案

If you're going to mix C, Python, and threads together, you need to make sure you properly initialize and manage Python's global interpreter lock (GIL). To do this, include the following code somewhere in your C code and make sure it's called prior to creation of any threads:

```
#include <Python.h>

... if (!PyEval_ThreadsInitialized()) {

    PyEval_InitThreads();
```

For any C code that involves Python objects or the Python C API, make sure you properly acquire and release the GIL first. This is done using `PyGILState_Ensure()` and `PyGILState_Release()`, as shown in the following:

```
.../* Make sure we own the GIL */ PyGILState_STATE state = PyGILState_Ensure();

/* Use functions in the interpreter / ... / Restore previous GIL state and return */
PyGILState_Release(state); ...
```

Every call to `PyGILState_Ensure()` must have a matching call to `PyGILState_Release()`.

讨论

In advanced applications involving C and Python, it is not uncommon to have many things going on at once—possibly involving a mix of a C code, Python code, C threads, and Python threads. As long as you diligently make sure the interpreter is properly initialized and that C code involving the interpreter has the proper GIL management calls, it all should work. Be aware that the `PyGILState_Ensure()` call does not immediately preempt or interrupt the interpreter. If other code is currently executing, this function will block until that code decides to release the GIL. Internally, the interpreter performs periodic thread switching, so even if another thread is executing, the caller will eventually get to run (although it may have to wait for a while first).

15.9 用WSIG包装C代码

问题

You have existing C code that you would like to access as a C extension module. You would like to do this using the Swig wrapper generator.

解决方案

Swig operates by parsing C header files and automatically creating extension code. To use it, you first need to have a C header file. For example, this header file for our sample code:

```
/* sample.h */

#include <math.h> extern int gcd(int, int); extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder); extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

Once you have the header files, the next step is to write a Swig “interface” file. By convention, these files have a `.i` suffix and might look similar to the following:

```
// sample.i - Swig interface %module sample %{ #include "sample.h" %}

/* Customizations */ %extend Point {
```

```
/* Constructor for Point objects */ Point(double x, double y) {
```

```
    Point *p = (Point *) malloc(sizeof(Point)); p->x = x; p->y = y; return p;
```

```
};
```

```
};
```

```
/* Map int *remainder as an output argument */ %include typemaps.i %apply int *OUTPUT {  
int * remainder };
```

```
/* Map the argument pattern (double *a, int n) to arrays */ %typemap(in) (double *a, int n)  
(Py_buffer view) {
```

```
    view.obj = NULL; if (PyObject_GetBuffer($input, &view, PyBUF_ANY_CONTIGUOUS |  
    PyBUF_FORMAT) == -1) {
```

```
        SWIG_fail;
```

```
    } if (strcmp(view.format, "d") != 0) {
```

```
        PyErr_SetString(PyExc_TypeError, "Expected an array of doubles"); SWIG_fail;
```

```
    } $1 = (double *) view.buf; $2 = view.len / sizeof(double);
```

```
}
```

```
%typemap(freearg) (double *a, int n) {  
    if (view$argnum.obj) {  
        PyBuffer_Release(&view$argnum);
```

```
    }
```

```
}
```

```
/* C declarations to be included in the extension module */
```

```
extern int gcd(int, int); extern int in_mandel(double x0, double y0, int n); extern int  
divide(int a, int b, int *remainder); extern double avg(double *a, int n);
```

```
typedef struct Point {  
    double x,y;
```

```
} Point;
```

```
extern double distance(Point *p1, Point *p2);
```

Once you have written the interface file, Swig is invoked as a command-line tool:

```
bash % swig -python -py3 sample.i bash %
```

The output of swig is two files, `sample_wrap.c` and `sample.py`. The latter file is what users import. The `sample_wrap.c` file is C code that needs to be compiled into a supporting module called `_sample`. This is done using the same techniques as for normal extension modules. For example, you create a `setup.py` file like this:

```
# setup.py from distutils.core import setup, Extension

setup(name='sample',
      py_modules=['sample.py'], ext_modules=[
          Extension('_sample',
                  ['sample_wrap.c'], include_dirs = [], define_macros = [],
                  undef_macros = [], library_dirs = [], libraries = ['sample'])
      ]
)
```

To compile and test, run `python3` on the `setup.py` file like this:

```
bash % python3 setup.py build_ext -inplace running build_ext building '_sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -
I/usr/local/include/python3.3m -c sample_wrap.c
```

```
-o build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o
```

```
sample_wrap.c: In function 'SWIG_InitializeModule': sample_wrap.c:3589: warning:
statement with no effect gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-
x86_64-3.3/sample.o
```

```
build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o -o _sample.so -lsample
```

```
bash %
```

If all of this works, you'll find that you can use the resulting C extension module in a straightforward way. For example:

```
>>> import sample
>>> sample.gcd(42,8)
2
>>> sample.divide(42,8)
[5, 2]
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
2.8284271247461903
>>> p1.x
2.0
>>> p1.y
3.0
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>>
```

讨论

Swig is one of the oldest tools for building extension modules, dating back to Python Python. Swig can automate much of the wrapper generation process.

All Swig interfaces tend to start with a short preamble like this:

```
%module sample %{ #include "sample.h" %}
```

This merely declares the name of the extension module and specifies C header files that must be included to make everything compile (the code enclosed in `%{` and `%}` is pasted directly into the output code so this is where you put all included files and other definitions needed for compilation). The bottom part of a Swig interface is a listing of C declarations that you want to be included in the extension. This is often just copied from the header files. In our example, we just pasted in the header file directly like this:

```
%module sample %{ #include "sample.h" %} ... extern int gcd(int, int); extern int
in_mandel(double x0, double y0, int n); extern int divide(int a, int b, int *remainder); extern
double avg(double *a, int n);
```

```
typedef struct Point {
    double x,y;
```

```
} Point;
```

```
extern double distance(Point *p1, Point *p2);
```

It is important to stress that these declarations are telling Swig what you want to include in the Python module. It is quite common to edit the list of declarations or to make modifications as appropriate. For example, if you didn't want certain declarations to be included, you would remove them from the declaration list. The most complicated part of using Swig is the various customizations that it can apply to the C code. This is a huge topic that can't be covered in great detail here, but a number of such customizations are shown in this recipe. The first customization involving the `%extend` directive allows methods to be attached to existing structure and class definitions. In the example, this is used to add a `con` - `structor` method to the `Point` structure. This customization makes it possible to use the structure like this:

```
>>> p1 = sample.Point(2,3)
>>>
```

If omitted, then `Point` objects would have to be created in a much more clumsy manner like this:

```
>>> # Usage if %extend Point is omitted
>>> p1 = sample.Point()
>>> p1.x = 2.0
>>> p1.y = 3
```

The second customization involving the inclusion of the `typemaps.i` library and the `%apply` directive is instructing Swig that the argument signature `int *remainder` is to be treated as an output value. This is actually a pattern matching rule. In all declarations that follow, any time `int *remainder` is encountered, it is handled as output. This customization is what makes the `divide()` function return two values:

```
>>> sample.divide(42,8)
[5, 2]
>>>
```

The last customization involving the `%typemap` directive is probably the most advanced feature shown here. A `typemap` is a rule that gets applied to specific argument patterns in the input. In this recipe, a `typemap` has been written to match the argument pattern (`double *a, int n`). Inside the `typemap` is a fragment of C code that tells Swig how to convert a Python object into the associated C arguments. The code in this recipe has been written using Python's buffer protocol in an attempt to match any input argument that looks like an array

of doubles (e.g., NumPy arrays, arrays created by the array module, etc.). See Recipe 15.3. Within the typemap code, substitutions such as \$1 and \$2 refer to variables that hold the converted values of the C arguments in the typemap pattern (e.g., \$1 maps to double *a and \$2 maps to int n). \$input refers to a PyObject * argument that was supplied as an input argument. \$argnum is the argument number. Writing and understanding typemaps is often the bane of programmers using Swig. Not only is the code rather cryptic, but you need to understand the intricate details of both the Python C API and the way in which Swig interacts with it. The Swig documentation has many more examples and detailed information. Nevertheless, if you have a lot of a C code to expose as an extension module, Swig can be a very powerful tool for doing it. The key thing to keep in mind is that Swig is basically a compiler that processes C declarations, but with a powerful pattern matching and customization component that lets you change the way in which specific declarations and types get processed. More information can be found at Swig’s website, including Python-specific documentation.

15.10 用Cython包装C代码

问题

You want to use Cython to make a Python extension module that wraps around an existing C library.

解决方案

Making an extension module with Cython looks somewhat similar to writing a hand-written extension, in that you will be creating a collection of wrapper functions. However, unlike previous recipes, you won’t be doing this in C—the code will look a lot more like Python. As preliminaries, assume that the sample code shown in the introduction to this chapter has been compiled into a C library called libsample. Start by creating a file named csample.pxd that looks like this:

```
# csample.pxd ## Declarations of “external” C functions and structures
```

```
cdef extern from “sample.h”:
```

```
    int gcd(int, int) bint in_mandel(double, double, int) int divide(int, int, int *) double  
    avg(double *, int) nogil
```

```
ctypedef struct Point:
```

```
    double x double y
```

```
    double distance(Point *, Point *)
```

This file serves the same purpose in Cython as a C header file. The initial declaration `cdef extern from "sample.h"` declares the required C header file. Declarations that follow are taken from that header. The name of this file is `csample.pxd`, not `sample.pxd`—this is important. Next, create a file named `sample.pyx`. This file will define wrappers that bridge the Python interpreter to the underlying C code declared in the `csample.pxd` file:

```
# sample.pyx
```

```
# Import the low-level C declarations cimport csample
```

```
# Import some functionality from Python and the C stdlib from cpython.pycapsule cimport *
```

```
from libc.stdlib cimport malloc, free
```

```
# Wrappers def gcd(unsigned int x, unsigned int y):
```

```
    return csample.gcd(x, y)
```

```
def in_mandel(x, y, unsigned int n):
```

```
    return csample.in_mandel(x, y, n)
```

```
def divide(x, y):
```

```
    cdef int rem quot = csample.divide(x, y, &rem) return quot, rem
```

```
def avg(double[:] a):
```

```
    cdef:
```

```
        int sz double result
```

```
    sz = a.size with nogil:
```

```
        result = csample.avg(<double *> &a[0], sz)
```

```
    return result
```

```
# Destructor for cleaning up Point objects cdef del_Point(object obj):
```

```
    pt = <csample.Point *> PyCapsule_GetPointer(obj, "Point") free(<void *> pt)
```

```
# Create a Point object and return as a capsule def Point(double x, double y):
```



```
cdef csample.Point *p p = <csample.Point *> malloc(sizeof(csample.Point)) if p == NULL:
```

```
    raise MemoryError("No memory to make a Point")
```

```
    p.x = x p.y = y return PyCapsule_New(<void *>p,"Point",  
<PyCapsule_Destructor>del_Point)
```

```
def distance(p1, p2):
```

```
    pt1 = <csample.Point *> PyCapsule_GetPointer(p1,"Point") pt2 = <csample.Point *>  
    PyCapsule_GetPointer(p2,"Point") return csample.distance(pt1,pt2)
```

Various details of this file will be covered further in the discussion section. Finally, to build the extension module, create a setup.py file that looks like this:

```
from distutils.core import setup from distutils.extension import Extension from  
Cython.Distutils import build_ext
```

```
ext_modules = [
```

```
    Extension('sample',
```

```
        ['sample.pyx'], libraries=['sample'], library_dirs=['.'])]
```

```
setup(
```

```
    name = 'Sample extension module', cmdclass = {'build_ext': build_ext}, ext_modules =  
    ext_modules
```

```
)
```

To build the resulting module for experimentation, type this:

```
bash % python3 setup.py build_ext -inplace running build_ext cythoning sample.pyx to  
sample.c building 'sample' extension gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -  
Wall -Wstrict-prototypes
```

```
-I/usr/local/include/python3.3m -c sample.c -o build/temp.macosx-10.6-x86_64-  
3.3/sample.o
```

```
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/sample.o  
-L. -lsample -o sample.so
```

```
bash %
```

If it works, you should have an extension module sample.so that can be used as shown in the following example:

```

>>> import sample
>>> sample.gcd(42,10)
2
>>> sample.in_mandel(1,1,400)
False
>>> sample.in_mandel(0,0,400)
True
>>> sample.divide(42,10)
(4, 2)
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1005d1e70>
>>> p2
<capsule object "Point" at 0x1005d1ea0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>

```

讨论

This recipe incorporates a number of advanced features discussed in prior recipes, including manipulation of arrays, wrapping opaque pointers, and releasing the GIL. Each of these parts will be discussed in turn, but it may help to review earlier recipes first. At a high level, using Cython is modeled after C. The .pxd files merely contain C definitions (similar to .h files) and the .pyx files contain implementation (similar to a .c file). The cimport statement is used by Cython to import definitions from a .pxd file. This is different than using a normal Python import statement, which would load a regular Python module. Although .pxd files contain definitions, they are not used for the purpose of automatically creating extension code. Thus, you still have to write simple wrapper functions. For example, even though the csample.pxd file declares `int gcd(int, int)` as a function, you still have to write a small wrapper for it in `sample.pyx`. For instance:

```
cimport csample
```

```
def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x,y)
```

For simple functions, you don't have to do too much. Cython will generate wrapper code that properly converts the arguments and return value. The C data types attached to the arguments are optional. However, if you include them, you get additional error checking for

free. For example, if someone calls this function with negative values, an exception is generated:

```
>>> sample.gcd(-10,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sample.pyx", line 7, in sample.gcd (sample.c:1284)
    def gcd(unsigned int x,unsigned int y):
OverflowError: can't convert negative value to unsigned int
>>>
```

If you want to add additional checking to the wrapper, just use additional wrapper code. For example:

```
def gcd(unsigned int x, unsigned int y):
    if x <= 0:
        raise ValueError("x must be > 0")

    if y <= 0:
        raise ValueError("y must be > 0")

    return csample.gcd(x,y)
```

The declaration of `in_mandel()` in the `csample.pxd` file has an interesting, but subtle definition. In that file, the function is declared as returning a `bint` instead of an `int`. This causes the function to create a proper Boolean value from the result instead of a simple integer. So, a return value of 0 gets mapped to `False` and 1 to `True`.

Within the Cython wrappers, you have the option of declaring C data types in addition to using all of the usual Python objects. The wrapper for `divide()` shows an example of this as well as how to handle a pointer argument.

```
def divide(x,y):
    cdef int rem quot = csample.divide(x,y,&rem) return quot, rem
```

Here, the `rem` variable is explicitly declared as a C `int` variable. When passed to the underlying `divide()` function, `&rem` makes a pointer to it just as in C. The code for the `avg()` function illustrates some more advanced features of Cython. First the declaration `def avg(double[:] a)` declares `avg()` as taking a one-dimensional memoryview of double values. The amazing part about this is that the resulting function will accept any compatible array object, including those created by libraries such as `numpy`. For example: `>>> import array >>> a = array.array('d',[1,2,3]) >>> import numpy >>> b = numpy.array([1., 2., 3.]) >>> import sample >>> sample.avg(a) 2.0 >>> sample.avg(b) 2.0 >>>`

In the wrapper, `a.size` and `&a[0]` refer to the number of array items and underlying pointer, respectively. The syntax `<double *> &a[0]` is how you type cast pointers to a different type if necessary. This is needed to make sure the C `avg()` receives a pointer of the correct type. Refer to the next recipe for some more advanced usage of Cython memoryviews. In addition to working with general arrays, the `avg()` example also shows how to work with the global interpreter lock. The statement with `nogil:` declares a block of code as executing without the GIL. Inside this block, it is illegal to work with any kind of normal Python object—only objects and functions declared as `cdef` can be used. In addition to that, external functions must explicitly declare that they can execute without the GIL. Thus, in the `csample.pxd` file, the `avg()` is declared as `double avg(double *, int) nogil`. The handling of the `Point` structure presents a special challenge. As shown, this recipe treats `Point` objects as opaque pointers using capsule objects, as described in Recipe 15.4. However, to do this, the underlying Cython code is a bit more complicated. First, the following imports are being used to bring in definitions of functions from the C library and Python C API:

```
from cpython.pycapsule cimport * from libc.stdlib cimport malloc, free
```

The function `del_Point()` and `Point()` use this functionality to create a capsule object that wraps around a `Point *` pointer. The declaration `cdef del_Point()` declares `del_Point()` as a function that is only accessible from Cython and not Python. Thus, this function will not be visible to the outside—instead, it’s used as a callback function to clean up memory allocated by the capsule. Calls to functions such as `PyCapsule_New()`, `PyCapsule_GetPointer()` are directly from the Python C API and are used in the same way. The `distance()` function has been written to extract pointers from the capsule objects created by `Point()`. One notable thing here is that you simply don’t have to worry about exception handling. If a bad object is passed, `PyCapsule_GetPointer()` raises an exception, but Cython already knows to look for it and propagate it out of the `distance()` function if it occurs. A downside to the handling of `Point` structures is that they will be completely opaque in this implementation. You won’t be able to peek inside or access any of their attributes. There is an alternative approach to wrapping, which is to define an extension type, as shown in this code:

```
# sample.pyx
```

```
cimport csample from libc.stdlib cimport malloc, free ...
```

```
cdef class Point:
```

```
    cdef csample.Point *_c_point def __cinit__(self, double x, double y):
```

```
        self._c_point = <csample.Point *> malloc(sizeof(csample.Point)) self._c_point.x = x
        self._c_point.y = y
```

```
    def __dealloc__(self):
```

```
        free(self._c_point)
```

```
    property x:
```

```
def __get__(self):
    return self._c_point.x
```

```
def __set__(self, value):
    self._c_point.x = value
```

property y:

```
def __get__(self):
    return self._c_point.y
```

```
def __set__(self, value):
    self._c_point.y = value
```

```
def distance(Point p1, Point p2):
    return csample.distance(p1._c_point, p2._c_point)
```

Here, the `cdef` class `Point` is declaring `Point` as an extension type. The class variable `csample.Point * _c_point` is declaring an instance variable that holds a pointer to an underlying `Point` structure in C. The `__cinit__()` and `__dealloc__()` methods create and destroy the underlying C structure using `malloc()` and `free()` calls. The property `x` and property `y` declarations give code that gets and sets the underlying structure attributes. The wrapper for `distance()` has also been suitably modified to accept instances of the `Point` extension type as arguments, but pass the underlying pointer to the C function. Making this change, you will find that the code for manipulating `Point` objects is more natural:

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<sample.Point object at 0x100447288>
>>> p2
<sample.Point object at 0x1004472a0>
>>> p1.x
2.0
>>> p1.y
3.0
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

This recipe has illustrated many of Cython's core features that you might be able to extrapolate to more complicated kinds of wrapping. However, you will definitely want to read more of the official documentation to do more. The next few recipes also illustrate a few additional Cython features.

15.11 用Cython写高性能的数组操作

问题

You would like to write some high-performance array processing functions to operate on arrays from libraries such as NumPy. You've heard that tools such as Cython can make this easier, but aren't sure how to do it.

解决方案

As an example, consider the following code which shows a Cython function for clipping the values in a simple one-dimensional array of doubles:

```
# sample.pyx (Cython)
```

```
cimport cython
```

```
@cython.boundscheck(False) @cython.wraparound(False) cpdef clip(double[:] a, double min, double max, double[:] out):
```

```
    """ Clip the values in a to be between min and max. Result in out """ if min > max:
```

```
        raise ValueError("min must be <= max")
```

```
    if a.shape[0] != out.shape[0]:
```

```
        raise ValueError("input and output arrays must be the same size")
```

```
    for i in range(a.shape[0]):
```

```
        if a[i] < min:
```

```
            out[i] = min
```

```
        elif a[i] > max:
```

```
            out[i] = max
```

```
        else:
```

```
            out[i] = a[i]
```

To compile and build the extension, you'll need a setup.py file such as the following (use `python3 setup.py build_ext -inplace` to build it):

```
from distutils.core import setup from distutils.extension import Extension from
Cython.Distutils import build_ext
```

```
ext_modules = [
    Extension('sample',
              ['sample.pyx'])
```

```
]
```

```
setup(  
    name = 'Sample app', cmdclass = {'build_ext': build_ext}, ext_modules = ext_modules  
)
```

You will find that the resulting function clips arrays, and that it works with many different kinds of array objects. For example:

```
>>> # array module example  
>>> import sample  
>>> import array  
>>> a = array.array('d', [1, -3, 4, 7, 2, 0])  
>>> a
```

```
array('d', [1.0, -3.0, 4.0, 7.0, 2.0, 0.0]) >>> sample.clip(a, 1, 4, a) >>> a  
array('d', [1.0, 1.0, 4.0, 4.0, 2.0, 1.0])
```

```
>>> # numpy example  
>>> import numpy  
>>> b = numpy.random.uniform(-10, 10, size=1000000)  
>>> b  
array([-9.55546017,  7.45599334,  0.69248932, ...,  0.69583148,  
       -3.86290931,  2.37266888])  
>>> c = numpy.zeros_like(b)  
>>> c  
array([ 0.,  0.,  0., ...,  0.,  0.,  0.])  
>>> sample.clip(b, -5, 5, c)  
>>> c  
array([-5.          ,  5.          ,  0.69248932, ...,  0.69583148,  
       -3.86290931,  2.37266888])  
>>> min(c)  
-5.0  
>>> max(c)  
5.0  
>>>
```

You will also find that the resulting code is fast. The following session puts our implementation in a head-to-head battle with the clip() function already present in numpy:

```
>>> timeit('numpy.clip(b,-5,5,c)', 'from __main__ import b,c,numpy', number=1000)
8.093049556000551
>>> timeit('sample.clip(b,-5,5,c)', 'from __main__ import b,c,sample',
...       number=1000)
3.760528204000366
>>>
```

As you can see, it's quite a bit faster—an interesting result considering the core of the NumPy version is written in C.

讨论

This recipe utilizes Cython typed memoryviews, which greatly simplify code that operates on arrays. The declaration `cpdef clip()` declares `clip()` as both a C-level and Python-level function. In Cython, this is useful, because it means that the function call is more efficiently called by other Cython functions (e.g., if you want to invoke `clip()` from a different Cython function). The typed parameters `double[:] a` and `double[:] out` declare those parameters as one-dimensional arrays of doubles. As input, they will access any array object that properly implements the memoryview interface, as described in PEP 3118. This includes arrays from NumPy and from the built-in array library.

When writing code that produces a result that is also an array, you should follow the convention shown of having an output parameter as shown. This places the responsibility of creating the output array on the caller and frees the code from having to know too much about the specific details of what kinds of arrays are being manipulated (it just assumes the arrays are already in-place and only needs to perform a few basic sanity checks such as making sure their sizes are compatible). In libraries such as NumPy, it is relatively easy to create output arrays using functions such as `numpy.zeros()` or `numpy.zeros_like()`. Alternatively, to create uninitialized arrays, you can use `numpy.empty()` or `numpy.empty_like()`. This will be slightly faster if you're about to overwrite the array contents with a result. In the implementation of your function, you simply write straightforward looking array processing code using indexing and array lookups (e.g., `a[i]`, `out[i]`, and so forth). Cython will take steps to make sure these produce efficient code. The two decorators that precede the definition of `clip()` are a few optional performance optimizations. `@cython.boundscheck(False)` eliminates all array bounds checking and can be used if you know the indexing won't go out of range. `@cython.wraparound(False)` eliminates the handling of negative array indices as wrapping around to the end of the array (like with Python lists). The inclusion of these decorators can make the code run substantially faster (almost 2.5 times faster on this example when tested). Whenever

working with arrays, careful study and experimentation with the underlying algorithm can also yield large speedups. For example, consider this variant of the clip() function that uses conditional expressions:

```
@cython.boundscheck(False) @cython.wraparound(False) cpdef clip(double[:] a, double min, double max, double[:] out):
```

```
    if min > max:
```

```
        raise ValueError("min must be <= max")
```

```
    if a.shape[0] != out.shape[0]:
```

```
        raise ValueError("input and output arrays must be the same size")
```

```
    for i in range(a.shape[0]):
```

```
        out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

When tested, this version of the code runs over 50% faster (2.44s versus 3.76s on the timeit() test shown earlier). At this point, you might be wondering how this code would stack up against a hand - written C version. For example, perhaps you write the following C function and craft a handwritten extension to using techniques shown in earlier recipes:

```
void clip(double *a, int n, double min, double max, double *out) {
    double x; for (; n >= 0; n--, a++, out++) {

        x = *a;

        *out = x > max ? max : (x < min ? min : x);

    }
}
```

The extension code for this isn't shown, but after experimenting, we found that a hand - crafted C extension ran more than 10% slower than the version created by Cython. The bottom line is that the code runs a lot faster than you might think. There are several extensions that can be made to the solution code. For certain kinds of array operations, it might make sense to release the GIL so that multiple threads can run in parallel. To do that, modify the code to include the with nogil: statement:

```
@cython.boundscheck(False) @cython.wraparound(False) cpdef clip(double[:] a, double min, double max, double[:] out):
```

```
if min > max:
```

```
    raise ValueError("min must be <= max")
```

```
if a.shape[0] != out.shape[0]:
```

```
    raise ValueError("input and output arrays must be the same size")
```

```
with nogil:
```

```
    for i in range(a.shape[0]):
```

```
        out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

If you want to write a version of the code that operates on two-dimensional arrays, here is what it might look like:

```
@cython.boundscheck(False) @cython.wraparound(False) cpdef clip2d(double[:,:] a, double min, double max, double[:,:] out):
```

```
    if min > max:
```

```
        raise ValueError("min must be <= max")
```

```
    for n in range(a.ndim):
```

```
        if a.shape[n] != out.shape[n]:
```

```
            raise TypeError("a and out have different shapes")
```

```
    for i in range(a.shape[0]):
```

```
        for j in range(a.shape[1]):
```

```
            if a[i,j] < min:
```

```
                out[i,j] = min
```

```
            elif a[i,j] > max:
```

```
                out[i,j] = max
```

```
            else:
```

```
                out[i,j] = a[i,j]
```

Hopefully it's not lost on the reader that all of the code in this recipe is not tied to any specific array library (e.g., NumPy). That gives the code a great deal of flexibility. However, it's also worth noting that dealing with arrays can be significantly more complicated once multiple dimensions, strides, offsets, and other factors are introduced. Those topics are beyond the scope of this recipe, but more information can be found in PEP 3118. The Cython documentation on "typed memoryviews" is also essential reading.

15.12 将函数指针转换为可调用对象

问题

You have (somehow) obtained the memory address of a compiled function, but want to turn it into a Python callable that you can use as an extension function.

解决方案

The ctypes module can be used to create Python callables that wrap around arbitrary memory addresses. The following example shows how to obtain the raw, low-level address of a C function and how to turn it back into a callable object:

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary(None)
>>> # Get the address of sin() from the C math library
>>> addr = ctypes.cast(lib.sin, ctypes.c_void_p).value
>>> addr
140735505915760
```

```
>>> # Turn the address into a callable function
>>> functype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)
>>> func = functype(addr)
>>> func
<CFunctionType object at 0x1006816d0>
```

```
>>> # Call the resulting function
>>> func(2)
0.9092974268256817
>>> func(0)
0.0
>>>
```

讨论

To make a callable, you must first create a CFUNCTYPE instance. The first argument to CFUNCTYPE() is the return type. Subsequent arguments are the types of the arguments. Once you have defined the function type, you wrap it around an integer memory address to create a callable object. The resulting object is used like any normal function accessed through ctypes. This recipe might look rather cryptic and low level. However, it is becoming increasingly common for programs and libraries to utilize advanced code generation

techniques like just in-time compilation, as found in libraries such as LLVM. For example, here is a simple example that uses the `llvmpy` extension to make a small assembly function, obtain a function pointer to it, and turn it into a Python callable:

```
>>> from llvm.core import Module, Function, Type, Builder
>>> mod = Module.new('example')
>>> f = Function.new(mod, Type.function(Type.double(), \
    [Type.double(), Type.double()], False), 'foo')
>>> block = f.append_basic_block('entry')
>>> builder = Builder.new(block)
>>> x2 = builder.fmul(f.args[0], f.args[0])
>>> y2 = builder.fmul(f.args[1], f.args[1])
>>> r = builder.fadd(x2, y2)
>>> builder.ret(r)
<llvm.core.Instruction object at 0x10078e990>
>>> from llvm.ee import ExecutionEngine
>>> engine = ExecutionEngine.new(mod)
>>> ptr = engine.get_pointer_to_function(f)
>>> ptr
4325863440
>>> foo = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double, ctypes.c_double)(ptr)
```

```
>>> # Call the resulting function
>>> foo(2,3)
13.0
>>> foo(4,5)
41.0
>>> foo(1,2)
5.0
>>>
```

It goes without saying that doing anything wrong at this level will probably cause the Python interpreter to die a horrible death. Keep in mind that you're directly working with machine-level memory addresses and native machine code—not Python functions.

15.13 传递NULL结尾的字符串给C函数库

问题

You are writing an extension module that needs to pass a NULL-terminated string to a C library. However, you're not entirely sure how to do it with Python's Unicode string implementation.

解决方案

Many C libraries include functions that operate on NULL-terminated strings declared as type `char *`. Consider the following C function that we will use for the purposes of illustration and testing:

```
void print_chars(char *s) {
    while (*s) {
        printf("%2x ", (unsigned char) *s);

        s++;

    } printf("\n");
}
```

This function simply prints out the hex representation of individual characters so that the passed strings can be easily debugged. For example: `print_chars("Hello");` // Outputs: 48 65 6c 6c 6f

For calling such a C function from Python, you have a few choices. First, you could restrict it to only operate on bytes using "y" conversion code to `PyArg_ParseTuple()` like this:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;

    if (!PyArg_ParseTuple(args, "y", &s)) {
        return NULL;

    } print_chars(s); Py_RETURN_NONE;
}
```

The resulting function operates as follows. Carefully observe how bytes with embedded NULL bytes and Unicode strings are rejected:

```
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be bytes without null bytes, not bytes
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
>>>
```

If you want to pass Unicode strings instead, use the “s” format code to PyArg_ParseTuple() such as this:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;

    if (!PyArg_ParseTuple(args, "s", &s)) {
        return NULL;
    }

    print_chars(s); Py_RETURN_NONE;
}
```

When used, this will automatically convert all strings to a NULL-terminated UTF-8 encoding. For example:

```
>>> print_chars('Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars('Spicy Jalape\u00f1o') # Note: UTF-8 encoding
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_chars('Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str without null characters, not str
>>> print_chars(b'Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>>
```

If for some reason, you are working directly with a PyObject* and can't use PyArg_ParseTuple(), the following code samples show how you can check and extract a suitable char* reference, from both a bytes and string object:

```
/* Some Python Object (obtained somehow) */ PyObject *obj;
```

```
/* Conversion from bytes */ {
```

```
    char *s; s = PyBytes_AsString(o); if (!s) {
```

```
        return NULL; /* TypeError already raised */
```

```
    } print_chars(s);
```

```
}
```

```
/* Conversion to UTF-8 bytes from a string */{
```

```
PyObject *bytes; char *s; if (!PyUnicode_Check(obj)) {
```

```
    PyErr_SetString(PyExc_TypeError, "Expected string"); return NULL;
```

```
    } bytes = PyUnicode_AsUTF8String(obj); s = PyBytes_AsString(bytes); print_chars(s);  
    Py_DECREF(bytes);
```

```
}
```

Both of the preceding conversions guarantee NULL-terminated data, but they do not check for embedded NULL bytes elsewhere inside the string. Thus, that's something that you would need to check yourself if it's important.

讨论

If it all possible, you should try to avoid writing code that relies on NULL-terminated strings since Python has no such requirement. It is almost always better to handle strings using the combination of a pointer and a size if possible. Nevertheless, sometimes you have to work with legacy C code that presents no other option. Although it is easy to use, there is a hidden memory overhead associated with using the "s" format code to `PyArg_ParseTuple()` that is easy to overlook. When you write code that uses this conversion, a UTF-8 string is created and permanently attached to the original string object. If the original string contains non-ASCII characters, this makes the size of the string increase until it is garbage collected. For example:

```
>>> import sys  
>>> s = 'Spicy Jalape\u00f1o'  
>>> sys.getsizeof(s)  
87  
>>> print_chars(s)      # Passing string  
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f  
>>> sys.getsizeof(s)   # Notice increased size  
103  
>>>
```

If this growth in memory use is a concern, you should rewrite your C extension code to use the `PyUnicode_AsUTF8String()` function like this:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {  
    PyObject *o, *bytes; char *s;
```

```

if (!PyArg_ParseTuple(args, "U", &o)) {
    return NULL;

} bytes = PyUnicode_AsUTF8String(o); s = PyBytes_AsString(bytes); print_chars(s);
Py_DECREF(bytes); Py_RETURN_NONE;

}

```

With this modification, a UTF-8 encoded string is created if needed, but then discarded after use. Here is the modified behavior:

```

>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
87
>>>

```

If you are trying to pass NULL-terminated strings to functions wrapped via ctypes, be aware that ctypes only allows bytes to be passed and that it does not check for embedded NULL bytes. For example:

```

>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary("./libsampl.e.so")
>>> print_chars = lib.print_chars
>>> print_chars.argtypes = (ctypes.c_char_p,)
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
48 65 6c 6c 6f
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <class 'TypeError'>: wrong type
>>>

```

If you want to pass a string instead of bytes, you need to perform a manual UTF-8 encoding first. For example:

```

>>> print_chars('Hello World'.encode('utf-8'))
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>>

```


For other extension tools (e.g., Swig, Cython), careful study is probably in order should you decide to use them to pass strings to C code.

15.14 传递Unicode字符串给C函数库

问题

You are writing an extension module that needs to pass a Python string to a C library function that may or may not know how to properly handle Unicode.

解决方案

There are many issues to be concerned with here, but the main one is that existing C libraries won't understand Python's native representation of Unicode. Therefore, your challenge is to convert the Python string into a form that can be more easily understood by C libraries. For the purposes of illustration, here are two C functions that operate on string data and output it for the purposes of debugging and experimentation. One uses bytes provided in the form `char *`, `int`, whereas the other uses wide characters in the form `wchar_t *`, `int`:

```
void print_chars(char *s, int len) {
    int n = 0;

    while (n < len) {
        printf("%2x ", (unsigned char) s[n]); n++;
    } printf("n");
}
```

```
void print_wchars(wchar_t *s, int len) {
    int n = 0; while (n < len) {

        printf("%x ", s[n]); n++;

    } printf("n");
}
```

For the byte-oriented function `print_chars()`, you need to convert Python strings into a suitable byte encoding such as UTF-8. Here is a sample extension function that does this:

```

static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s; Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "s#", &s, &len)) {
        return NULL;

    } print_chars(s, len); Py_RETURN_NONE;

}

```

For library functions that work with the machine native `wchar_t` type, you can write extension code such as this:

```

static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    wchar_t *s; Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "u#", &s, &len)) {
        return NULL;

    } print_wchars(s, len); Py_RETURN_NONE;

}

```

Here is an interactive session that illustrates how these functions work:

```

>>> s = 'Spicy Jalape\u00f1o'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>>

```

Carefully observe how the byte-oriented function `print_chars()` is receiving UTF-8 encoded data, whereas `print_wchars()` is receiving the Unicode code point values.

讨论

Before considering this recipe, you should first study the nature of the C library that you're accessing. For many C libraries, it might make more sense to pass bytes instead of a string. To do that, use this conversion code instead:

```

static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s; Py_ssize_t len;

```

```

/* accepts bytes, bytearray, or other byte-like object */ if (!PyArg_ParseTuple(args, "y#",
&s, &len)) {

    return NULL;

} print_chars(s, len); Py_RETURN_NONE;

}

```

If you decide that you still want to pass strings, you need to know that Python 3 uses an adaptable string representation that is not entirely straightforward to map directly to C libraries using the standard types `char *` or `wchar_t *`. See PEP 393 for details. Thus, to present string data to C, some kind of conversion is almost always necessary. The `s#` and `u#` format codes to `PyArg_ParseTuple()` safely perform such conversions. One potential downside is that such conversions cause the size of the original string object to permanently increase. Whenever a conversion is made, a copy of the converted data is kept and attached to the original string object so that it can be reused later. You can observe this effect:

```

>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
103
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>> sys.getsizeof(s)
163
>>>

```

For small amounts of string data, this might not matter, but if you're doing large amounts of text processing in extensions, you may want to avoid the overhead. Here is an alternative implementation of the first extension function that avoids these memory inefficiencies:

```

static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    PyObject *obj, *bytes; char *s; Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;

    } bytes = PyUnicode_AsUTF8String(obj); PyBytes_AsStringAndSize(bytes, &s, &len);
    print_chars(s, len); Py_DECREF(bytes); Py_RETURN_NONE;
}

```

```
}
```

Avoiding memory overhead for `wchar_t` handling is much more tricky. Internally, Python stores strings using the most efficient representation possible. For example, strings containing nothing but ASCII are stored as arrays of bytes, whereas strings containing characters in the range U+0000 to U+FFFF use a two-byte representation. Since there isn't a single representation of the data, you can't just cast the internal array to `wchar_t *` and hope that it works. Instead, a `wchar_t` array has to be created and text copied into it. The "u#" format code to `PyArg_ParseTuple()` does this for you at the cost of efficiency (it attaches the resulting copy to the string object). If you want to avoid this long-term memory overhead, your only real choice is to copy the Unicode data into a temporary array, pass it to the C library function, and then deallocate the array. Here is one possible implementation:

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    PyObject *obj; wchar_t *s; Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;

    } if ((s = PyUnicode_AsWideCharString(obj, &len)) == NULL) {

        return NULL;

    } print_wchars(s, len); PyMem_Free(s); Py_RETURN_NONE;
}
```

In this implementation, `PyUnicode_AsWideCharString()` creates a temporary buffer of `wchar_t` characters and copies data into it. That buffer is passed to C and then released afterward. As of this writing, there seems to be a possible bug related to this behavior, as described at the Python issues page.

If, for some reason you know that the C library takes the data in a different byte encoding than UTF-8, you can force Python to perform an appropriate conversion using extension code such as the following:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s = 0; int len; if (!PyArg_ParseTuple(args, "es#", "encoding-name", &s, &len)) {

        return NULL;

    } print_chars(s, len); PyMem_Free(s); Py_RETURN_NONE;
}
```

Last, but not least, if you want to work directly with the characters in a Unicode string, here is an example that illustrates low-level access:

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    PyObject *obj; int n, len; int kind; void *data;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;

    } if (PyUnicode_READY(obj) < 0) {

        return NULL;

    }

    len = PyUnicode_GET_LENGTH(obj); kind = PyUnicode_KIND(obj); data =
    PyUnicode_DATA(obj);

    for (n = 0; n < len; n++) {
        Py_UCS4 ch = PyUnicode_READ(kind, data, n); printf("%x ", ch);

    } printf("n"); Py_RETURN_NONE;
}
```

In this code, the `PyUnicode_KIND()` and `PyUnicode_DATA()` macros are related to the variable-width storage of Unicode, as described in PEP 393. The `kind` variable encodes information about the underlying storage (8-bit, 16-bit, or 32-bit) and `data` points the buffer. In reality, you don't need to do anything with these values as long as you pass them to the `PyUnicode_READ()` macro when extracting characters. A few final words: when passing Unicode strings from Python to C, you should probably try to make it as simple as possible. If given the choice between an encoding such as

UTF-8 or wide characters, choose UTF-8. Support for UTF-8 seems to be much more common, less trouble-prone, and better supported by the interpreter. Finally, make sure you review the documentation on Unicode handling.

15.15 C字符串转换为Python字符串

问题

You want to convert strings from C to Python bytes or a string object.

解决方案

For C strings represented as a pair `char *`, `int`, you must decide whether or not you want the string presented as a raw byte string or as a Unicode string. Byte objects can be built using `Py_BuildValue()` as follows:

```
char s; / Pointer to C string data / int len; / Length of data */
```

```
/* Make a bytes object */ PyObject *obj = Py_BuildValue("y#", s, len);
```

If you want to create a Unicode string and you know that `s` points to data encoded as UTF-8, you can use the following:

```
PyObject *obj = Py_BuildValue("s#", s, len);
```

If `s` is encoded in some other known encoding, you can make a string using `PyUnicode_Decode()` as follows:

```
PyObject *obj = PyUnicode_Decode(s, len, "encoding", "errors");
```

```
/* Examples */ obj = PyUnicode_Decode(s, len, "latin-1", "strict"); obj =  
PyUnicode_Decode(s, len, "ascii", "ignore");
```

If you happen to have a wide string represented as a `wchar_t *`, `len` pair, there are a few options. First, you could use `Py_BuildValue()` as follows:

```
wchar_t w; / Wide character string / int len; / Length */
```

```
PyObject *obj = Py_BuildValue("u#", w, len);
```

Alternatively, you can use `PyUnicode_FromWideChar()`:

```
PyObject *obj = PyUnicode_FromWideChar(w, len);
```

For wide character strings, no interpretation is made of the character data—it is assumed to be raw Unicode code points which are directly converted to Python.

讨论

Conversion of strings from C to Python follow the same principles as I/O. Namely, the data from C must be explicitly decoded into a string according to some codec. Common encodings include ASCII, Latin-1, and UTF-8. If you're not entirely sure of the encoding or the data is binary, you're probably best off encoding the string as bytes instead. When

making an object, Python always copies the string data you provide. If necessary, it's up to you to release the C string afterward (if required). Also, for better reliability, you should try to create strings using both a pointer and a size rather than relying on NULL-terminated data.

15.16 不确定编码格式的C字符串

问题

You are converting strings back and forth between C and Python, but the C encoding is of a dubious or unknown nature. For example, perhaps the C data is supposed to be UTF-8, but it's not being strictly enforced. You would like to write code that can handle malformed data in a graceful way that doesn't crash Python or destroy the string data in the process.

解决方案

Here is some C data and a function that illustrates the nature of this problem:

```
/* Some dubious string data (malformed UTF-8) */ const char *sdata = "Spicy  
Jalapexc3xb1oxae"; int slen = 16;
```

```
/* Output character data */ void print_chars(char *s, int len) {
```

```
    int n = 0; while (n < len) {
```

```
        printf("%2x ", (unsigned char) s[n]); n++;
```

```
    } printf("n");
```

```
}
```

In this code, the string `sdata` contains a mix of UTF-8 and malformed data. Nevertheless, if a user calls `print_chars(sdata, slen)` in C, it works fine. Now suppose you want to convert the contents of `sdata` into a Python string. Further suppose you want to later pass that string to the `print_chars()` function through an extension. Here's how to do it in a way that exactly preserves the original data even though there are encoding problems:

```
/* Return the C string back to Python */ static PyObject *py_retstr(PyObject *self, PyObject  
*args) {
```

```

if (!PyArg_ParseTuple(args, "")) {
    return NULL;

} return PyUnicode_Decode(sdata, slen, "utf-8", "surrogateescape");

}

/* Wrapper for the print_chars() function */ static PyObject *py_print_chars(PyObject *self,
PyObject *args) {

    PyObject *obj, *bytes; char *s = 0; Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;

    }

    if ((bytes = PyUnicode_AsEncodedString(obj, "utf-8", "surrogateescape"))
        == NULL) {

        return NULL;

    } PyBytes_AsStringAndSize(bytes, &s, &len); print_chars(s, len); Py_DECREF(bytes);
    Py_RETURN_NONE;

}

```

If you try these functions from Python, here's what happens:

```

>>> s = retstr()
>>> s
'Spicy Jalapeño\uudcaé'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f ae
>>>

```

Careful observation will reveal that the malformed string got encoded into a Python string without errors, and that when passed back into C, it turned back into a byte string that exactly encoded the same bytes as the original C string.

讨论

This recipe addresses a subtle, but potentially annoying problem with string handling in extension modules. Namely, the fact that C strings in extensions might not follow the strict Unicode encoding/decoding rules that Python normally expects. Thus, it's possible that some malformed C data would pass to Python. A good example might be C strings associated with low-level system calls such as filenames. For instance, what happens if a system call returns a broken string back to the interpreter that can't be properly decoded.

Normally, Unicode errors are often handled by specifying some sort of error policy, such as strict, ignore, replace, or something similar. However, a downside of these policies is that they irreparably destroy the original string content. For example, if the malformed data in the example was decoded using one of these policies, you would get results such as this:

```
>>> raw = b'Spicy Jalape\xc3\xb1o\xae'
>>> raw.decode('utf-8', 'ignore')
'Spicy Jalapeño'
>>> raw.decode('utf-8', 'replace')
'Spicy Jalapeño?'
>>>
```

The surrogateescape error handling policy takes all nondecodable bytes and turns them into the low-half of a surrogate pair (udcXX where XX is the raw byte value). For example:

```
>>> raw.decode('utf-8', 'surrogateescape')
'Spicy Jalapeño\uudcae'
>>>
```

Isolated low surrogate characters such as `udcae` never appear in valid Unicode. Thus, this string is technically an illegal representation. In fact, if you ever try to pass it to functions that perform output, you'll get encoding errors:

```
>>> s = raw.decode('utf-8', 'surrogateescape')
>>> print(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\uudcae'
in position 14: surrogates not allowed
>>>
```

However, the main point of allowing the surrogate escapes is to allow malformed strings to pass from C to Python and back into C without any data loss. When the string is encoded using surrogateescape again, the surrogate characters are turned back into their original

bytes. For example:

```
>>> s
'Spicy Jalapeño\uudcaé'
>>> s.encode('utf-8', 'surrogateescape')
b'Spicy Jalape\xc3\xb1o\xae'
>>>
```

As a general rule, it's probably best to avoid surrogate encoding whenever possible— your code will be much more reliable if it uses proper encodings. However, sometimes there are situations where you simply don't have control over the data encoding and you aren't free to ignore or replace the bad data because other functions may need to use it. This recipe shows how to do it.

As a final note, many of Python's system-oriented functions, especially those related to filenames, environment variables, and command-line options, use surrogate encoding. For example, if you use a function such as `os.listdir()` on a directory containing an undecodable filename, it will be returned as a string with surrogate escapes. See Recipe 5.15 for a related recipe. PEP 383 has more information about the problem addressed by this recipe and surrogate escape error handling.

15.17 传递文件名给C扩展

问题

You need to pass filenames to C library functions, but need to make sure the filename has been encoded according to the system's expected filename encoding.

解决方案

To write an extension function that receives a filename, use code such as this:

```
static PyObject *py_get_filename(PyObject *self, PyObject *args) {
    PyObject *bytes; char *filename; Py_ssize_t len; if (!PyArg_ParseTuple(args, "O&",
    PyUnicode_FSConverter, &bytes)) {

        return NULL;

    } PyBytes_AsStringAndSize(bytes, &filename, &len); /* Use filename */ ...

    /* Cleanup and return */ Py_DECREF(bytes) Py_RETURN_NONE;
```

```
}
```

If you already have a `PyObject *` that you want to convert as a filename, use code such as the following:

```
PyObject obj; / Object with the filename */ PyObject *bytes; char *filename; Py_ssize_t len;

bytes = PyUnicode_EncodeFSDefault(obj); PyBytes_AsStringAndSize(bytes, &filename,
&len); /* Use filename */ ...

/* Cleanup */ Py_DECREF(bytes);
```

If you need to return a filename back to Python, use the following code:

```
/* Turn a filename into a Python object */

char filename; / Already set / int filename_len; / Already set */

PyObject *obj = PyUnicode_DecodeFSDefaultAndSize(filename, filename_len);
```

讨论

Dealing with filenames in a portable way is a tricky problem that is best left to Python. If you use this recipe in your extension code, filenames will be handled in a manner that is consistent with filename handling in the rest of Python. This includes encoding/ decoding of bytes, dealing with bad characters, surrogate escapes, and other complications.

15.18 传递已打开的文件给C扩展

问题

You have an open file object in Python, but need to pass it to C extension code that will use the file.

解决方案

To convert a file to an integer file descriptor, use `PyFile_FromFd()`, as shown:

```
PyObject fobj; / File object (already obtained somehow) */ int fd =
PyObject_AsFileDescriptor(fobj); if (fd < 0) {

    return NULL;

}
```

The resulting file descriptor is obtained by calling the `fileno()` method on `fobj`. Thus, any object that exposes a descriptor in this manner should work (e.g., file, socket, etc.). Once you have the descriptor, it can be passed to various low-level C functions that expect to work with files. If you need to convert an integer file descriptor back into a Python object, use `PyFile_FromFd()` as follows:

```
int fd; /* Existing file descriptor (already open) */ PyObject *fobj = PyFile_FromFd(fd,
"filename", "r", -1, NULL, NULL, NULL, 1);
```

The arguments to `PyFile_FromFd()` mirror those of the built-in `open()` function. `NULL` values simply indicate that the default settings for the encoding, errors, and newline arguments are being used.

讨论

If you are passing file objects from Python to C, there are a few tricky issues to be concerned about. First, Python performs its own I/O buffering through the `io` module. Prior to passing any kind of file descriptor to C, you should first flush the I/O buffers on the associated file objects. Otherwise, you could get data appearing out of order on the file stream. Second, you need to pay careful attention to file ownership and the responsibility of closing the file in particular. If a file descriptor is passed to C, but still used in Python, you need to make sure C doesn't accidentally close the file. Likewise, if a file descriptor is being turned into a Python file object, you need to be clear about who is responsible for closing it. The last argument to `PyFile_FromFd()` is set to 1 to indicate that Python should close the file. If you need to make a different kind of file object such as a `FILE *` object from the C standard I/O library using a function such as `fdopen()`, you'll need to be especially careful. Doing so would introduce two completely different I/O buffering layers into the I/O stack (one from Python's `io` module and one from C `stdio`). Operations such as `fclose()` in C could also inadvertently close the file for further use in Python. If given a choice, you should probably make extension code work with the low-level integer file descriptors as opposed to using a higher-level abstraction such as that provided by `<stdio.h>`.

15.19 从C语言中读取类文件对象

问题

You want to write C extension code that consumes data from any Python file-like object (e.g., normal files, StringIO objects, etc.).

解决方案

To consume data on a file-like object, you need to repeatedly invoke its `read()` method and take steps to properly decode the resulting data. Here is a sample C extension function that merely consumes all of the data on a file-like object and dumps it to standard output so you can see it:

```
#define CHUNK_SIZE 8192
```

```
/* Consume a "file-like" object and write bytes to stdout */ static PyObject
```

```
*py_consume_file(PyObject *self, PyObject *args) {
```

```
    PyObject *obj; PyObject *read_meth; PyObject *result = NULL; PyObject *read_args;
```

```
    if (!PyArg_ParseTuple(args, "O", &obj)) {
```

```
        return NULL;
```

```
    }
```

```
    /* Get the read method of the passed object */ if ((read_meth =  
    PyObject_GetAttrString(obj, "read")) == NULL) {
```

```
        return NULL;
```

```
    }
```

```
    /* Build the argument list to read() */ read_args = Py_BuildValue("(i)", CHUNK_SIZE);
```

```
    while (1) {
```

```

PyObject *data; PyObject *enc_data; char *buf; Py_ssize_t len;

/* Call read() */ if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {

    goto final;

}

/* Check for EOF */ if (PySequence_Length(data) == 0) {

    Py_DECREF(data); break;

}

/* Encode Unicode as Bytes for C */ if
((enc_data=PyUnicode_AsEncodedString(data,"utf-8","strict"))==NULL) {

    Py_DECREF(data); goto final;

}

/* Extract underlying buffer data */ PyBytes_AsStringAndSize(enc_data, &buf,
&len);

/* Write to stdout (replace with something more useful) */ write(1, buf, len);

/* Cleanup */ Py_DECREF(enc_data); Py_DECREF(data);

} result = Py_BuildValue("");

final:
/* Cleanup */ Py_DECREF(read_meth); Py_DECREF(read_args); return result;

}

```

To test the code, try making a file-like object such as a StringIO instance and pass it in:

```

>>> import io
>>> f = io.StringIO('Hello\nWorld\n')
>>> import sample
>>> sample.consume_file(f)
Hello
World
>>>

```

讨论

Unlike a normal system file, a file-like object is not necessarily built around a low-level file descriptor. Thus, you can't use normal C library functions to access it. Instead, you need to use Python's C API to manipulate the file-like object much like you would in Python. In the solution, the `read()` method is extracted from the passed object. An argument list is built and then repeatedly passed to `PyObject_Call()` to invoke the method. To detect end-of-file (EOF), `PySequence_Length()` is used to see if the returned result has zero length. For all I/O operations, you'll need to concern yourself with the underlying encoding and distinction between bytes and Unicode. This recipe shows how to read a file in text mode and decode the resulting text into a bytes encoding that can be used by C. If you want to read the file in binary mode, only minor changes will be made. For example:

```
...
/* Call read() */ if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
    goto final;
}

/* Check for EOF */ if (PySequence_Length(data) == 0) {
    Py_DECREF(data); break;
} if (!PyBytes_Check(data)) {
    Py_DECREF(data); PyErr_SetString(PyExc_IOError, "File must be in binary mode");
    goto final;
}

/* Extract underlying buffer data */ PyBytes_AsStringAndSize(data, &buf, &len); ...
```

The trickiest part of this recipe concerns proper memory management. When working with `PyObject *` variables, careful attention needs to be given to managing reference counts and cleaning up values when no longer needed. The various `Py_DECREF()` calls are doing this. The recipe is written in a general-purpose manner so that it can be adapted to other file operations, such as writing. For example, to write data, merely obtain the `write()` method of the file-like object, convert data into an appropriate Python object (bytes or Unicode), and invoke the method to have it written to the file. Finally, although file-like objects often

provide other methods (e.g., `readline()`, `read_into()`), it is probably best to just stick with the basic `read()` and `write()` methods for maximal portability. Keeping things as simple as possible is often a good policy for C extensions.

15.20 处理C语言中的可迭代对象

问题

You want to write C extension code that consumes items from any iterable object such as a list, tuple, file, or generator.

解决方案

Here is a sample C extension function that shows how to consume the items on an iterable:

```
static PyObject *py_consume_iterable(PyObject *self, PyObject *args) {
    PyObject *obj; PyObject *iter; PyObject *item;

    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    } if ((iter = PyObject_GetIter(obj)) == NULL) {
        return NULL;
    } while ((item = PyIter_Next(iter)) != NULL) {
        /* Use item */ ... Py_DECREF(item);
    }
    Py_DECREF(iter); return Py_BuildValue("");
}
```

讨论

The code in this recipe mirrors similar code in Python. The `PyObject_GetIter()` call is the same as calling `iter()` to get an iterator. The `PyIter_Next()` function invokes the next method on the iterator returning the next item or `NULL` if there are no more items. Make sure you're careful with memory management—`Py_DECREF()` needs to be called on both the produced items and the iterator object itself to avoid leaking memory.

15.21 诊断分析代码错误

问题

The interpreter violently crashes with a segmentation fault, bus error, access violation, or other fatal error. You would like to get a Python traceback that shows you where your program was running at the point of failure.

解决方案

The `faulthandler` module can be used to help you solve this problem. Include the following code in your program:

```
import faulthandler faulthandler.enable()
```

Alternatively, run Python with the `-Xfaulthandler` option such as this:

```
bash % python3 -Xfaulthandler program.py
```

Last, but not least, you can set the `PYTHONFAULTHANDLER` environment variable. With `faulthandler` enabled, fatal errors in C extensions will result in a Python trace - back being printed on failures. For example:

```
Fatal Python error: Segmentation fault
```

```
Current thread 0x00007fff71106cc0:
```

```
File "example.py", line 6 in foo File "example.py", line 10 in bar File "example.py", line 14 in spam File "example.py", line 19 in <module>
```

```
Segmentation fault
```

Although this won't tell you where in the C code things went awry, at least it can tell you how it got there from Python.

讨论

The `faulthandler` will show you the stack traceback of the Python code executing at the time of failure. At the very least, this will show you the top-level extension function that was invoked. With the aid of `pdb` or other Python debugger, you can investigate the flow of the

Python code leading to the error. `faulthandler` will not tell you anything about the failure from C. For that, you will need to use a traditional C debugger, such as `gdb`. However, the information from the `faulthandler` traceback may give you a better idea of where to direct your attention. It should be noted that certain kinds of errors in C may not be easily recoverable. For example, if a C extension trashes the stack or program heap, it may render `faulthandler` inoperable and you'll simply get no output at all (other than a crash). Obviously, your mileage may vary.

附录A

在线资源

<http://docs.python.org>

如果你需要深入了解探究语言和模块的细节，那么不必说，Python自家的在线文档是一个卓越的资源。只要保证你查看的是python 3的文档而不是以前的老版本

<http://www.python.org/dev/peps>

如果你向理解为python语言添加新特性的动机以及实现的细节，那么PEPs（Python Enhancement Proposals—Python开发编码规范）绝对是非常宝贵的资源。尤其是一些高级语言功能更是如此。在写这本书的时候，PEPS通常比官方文档管用。

<http://pyvideo.org>

这里有来自最近的PyCon大会、用户组见面会等的大量视频演讲和教程素材。对于学习潮流的python开发是非常宝贵的资源。许多视频中都会有Python的核心开发者现身说法，讲解Python 3中添加的新特性。

<http://code.activestate.com/recipes/langs/python>

长期以来，ActiveState的Python版块已经成为一个找到数以千计的针对特定编程问题的解决方案。到写作此书位置，已经包含了大约300个特定于Python3的秘籍。你会发现，其中多数的秘籍要么对本书覆盖的话题进行了扩展，要么专精于具体的任务。所以说，它是一个好伴侣。

<http://stackoverflow.com/questions/tagged/python>

Stack Overflow 木器啊有超过175,000个问题呗标记为Python相关（而其中大约5000个问题是针对Python 3的）。尽管问题和回答的质量不同，但是任仍然能发现很多好优秀的素材。

Python学习书籍

下面这些书籍提供了对Python编程的入门介绍，且重点放在了Python 3上。

Beginning Python: From Novice to Professional, 2nd Edition, by Magnus Lie Hetland, Apress (2008). Programming in Python 3, 2nd Edition, by Mark Summerfield, Addison-Wesley (2010).

- Learning Python, 第四版，作者 Mark Lutz, O'Reilly & Associates 出版 (2009)。
- The Quick Python Book, 作者 Vernon Ceder, Manning 出版(2010)。
- Python Programming for the Absolute Beginner, 第三版，作者 Michael Dawson, Course Technology PTR 出版(2010)。
- Beginning Python: From Novice to Professional, 第二版，作者 Magnus Lie Hetland, Apress 出版(2008)。
- Programming in Python 3, 第二版，作者 Mark Summerfield, Addison-Wesley 出版 (2010)。

高级书籍

下面的这些书籍提供了更多高级的范围，也包含Python 3方面的内容。

- Programming Python, 第四版, by Mark Lutz, O'Reilly & Associates 出版(2010)。
- Python Essential Reference, 第四版，作者 David Beazley, Addison-Wesley 出版 (2009)。
- Core Python Applications Programming, 第三版，作者 Wesley Chun, Prentice Hall 出版(2012)。
- The Python Standard Library by Example, 作者 Doug Hellmann, Addison-Wesley 出版(2011)。
- Python 3 Object Oriented Programming, 作者 Dusty Phillips, Packt Publishing 出版 (2010)。
- Porting to Python 3, 作者 Lennart Regebro, CreateSpace 出版(2011), <http://python3porting.com>.

关于译者

关于译者

- 姓名：熊能
- Email: yidao620@gmail.com
- 博客: <http://yidao620c.github.io/>
- GitHub: <https://github.com/yidao620c>

主要贡献者

1. MoguCloud (<https://github.com/MoguCloud>)
2. littlezz (zz.at.field@gmail.com)
3. xiaotiaobu (<https://github.com/xiaotiaobu>)
4. Eskibear (<https://github.com/Eskibear>)
5. LiHaoGit (dahao647@gmail.com)
6. Jason Chang (crazycashier@icoud.com)
7. Yu Longjun (<https://github.com/yulongjun>)
8. hanxleon (leonhanxl@gmail.com)
9. slideclick (<https://github.com/slideclick>)
10. Tony Yang (liuliu036@gmail.com)
11. nivance (<https://github.com/nivance>)

项目主页

<https://github.com/yidao620c/python3-cookbook>

Roadmap

2014/08/10 - 2014/08/31:

- | `github`项目搭建, `readthedocs`文档生成。
- | 整个项目的框架完成

2014/09/01 - 2014/10/31:

- | 前4章翻译完成

2014/11/01 - 2015/01/31:

- | 前8章翻译完成

2015/02/01 - 2015/03/31:

| 前9章翻译完成

2015/04/01 - 2015/05/31:

| 10章翻译完成

2015/06/01 - 2015/06/30:

| 11章翻译完成

2015/07/01 - 2015/07/31:

| 12章翻译完成

2015/08/01 - 2015/08/31:

| 13章翻译完成

2015/09/01 - 2015/09/30:

| 14章翻译完成

2015/10/01 - 2015/10/31:

| 15章翻译完成

2015/11/01 - 2015/11/15:

| 对全部翻译进行校对一次

2015/11/16 - 2015/11/20:

| 对外公开发布完整版1.0，包括转换后的PDF文件